Politecnico di Torino

Master's Degree Thesis

Portfolio management and Deep learning: Reinforcement learning and Transformer applied to stock market data

Supervisors

Prof. Enrico BIBBONA

Prof. Patrizia SEMERARO

Candidate

Marco GULLOTTO

Master's Degree in Data Science and engineering

10 2021

Summary

This thesis, developed during an internship in Add-For S.p.A. and Gamma Capital Markets Ltd., is a research thesis in the Fintech field to design a new financial strategy in "Portfolio management and selection". The task is to make investment decisions based on strategies that ensure maximum profit for each investment period. It is a complicated optimization problem that aims to find the best actions to select the most profitable assets in a predefined period. This task is challenging due to the difficulties in representing asset price series as these are non-stationary and have noise and fluctuations. An asset is an item with a certain economic value that can be bought, sold or rented. In this thesis, we consider only cash and stock as assets. The latter are registered titles that represent the participation in a firm's equity. To manage the stationary and reduce the noise in the input data, the concept of return is introduced. A return, also known as a financial return, in its simplest terms, is the money earned or lost on an investment over a certain period of time. Deep learning has been used to address this problem, but the final results are rather poor and not applicable to real problems. The goal of this thesis is to study the state of the art in the field of portfolio management and selection, with particular emphasis on the use of Transformer architecture and reinforcement learning (RL), and the application of these models to real problems. An in-depth analysis was conducted both to choose the best machine learning (ML) architecture and the most suitable RL model. The concept of "Inductive Bias" was exploited to study the strengths and weaknesses of the most common ML networks such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Transformers. It turns out that all of these architectures show different advantages and some drawbacks, with the Transformer showing better behaviour and reliability concerning the portfolio optimization problem. The transformer network is relatively new and relies primarily on the attention mechanism, which is used in place of recurrent or convolutional layers to learn both long and short-term dependencies. The attention mechanism was introduced in the context of natural language processing, but can easily be generalized to other types of data such as time series. From the version of the standard transformer, introduced by Vaswani et al. other variants have been introduced to increase

performance in various settings. Although this architecture is very performing, it is very complex to tune and more problems arise when trying to apply it to RL algorithms. For this reason, Parisotto et al. have proposed a new variant of the Transformer which greatly increases the convergence speed and stability when applied to RL problems. This solution is widely accepted by the community and seems to perform well in different situations and problems. Other proposed variants look promising, but their results have not been validated unlike the work done by Parisotto. From the literature, several RL algorithms can be used for this type of task. Due to the high stochasticity of the model environment, only a model-free approach can be used. This means we do not know the whole dynamics and reward function of that Markov Decision Process. Hence, model-free RL ignores the model and cares less about the inner workings. It uses sampling and simulation to understand how to maximize the cumulative reward. Starting from previous Fintech works, many solutions foresee the use of DDPG (deep deterministic policy gradient) proposed by Lillicrap et al. This algorithm can be seen as the extension of Deep Q-Network (DQN) to continuous action spaces. So, in the first experiment to solve the portfolio management task, the agent basically follows the DDPG structure using Parisotto Transformer encoders whose architecture is robust to long-term dependencies and partial observability. Once the training is complete, by examining different time series of assets, the agent is able to predict the best asset or set of assets in which to invest. However, DDPG is a pretty old algorithm, and more up-to-date solutions are available. For this reason, other RL algorithms are tested to see if newer solutions could improve the baseline obtained with DDPG. The most significant improvement is obtained with the Soft Actor-Critic (SAC) proposed by Haarnoja et al.. SAC is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and the DDPG-style approaches. A central feature of this algorithm is the so-called "entropy regulation". The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. In this way, our model can design more original and profitable solutions than the previous DDPG-based approach. A significant amount of time is also devoted to redesigning the Markov Decision Problem. The solutions used in the literature tend to underestimate the complexity of this task and this has led the RL agent to select a suboptimal solution. The results outperform the current state of the art, but the final method does not seem stable enough to be used in a real-case scenario. The proposed algorithm appears to be unstable as we increase the number of assets considered. Several analyses were carried out to understand these behaviours and various solutions were proposed (for future work).

Acknowledgements

"The only limit is the one you set yourself."
Dedicato a tutti coloro che si sono presi cura di me.

Table of Contents

List of Figures

Chapter 1

Portfolio management and Selection

1.1 Introduction

Portfolio optimization is the process of selecting the best portfolio (asset distribution), out of the set of all portfolios being considered, according to some objective [1] [2]. Our portfolio can be constituted of several tools: one tool is preferred over the other depending on the level of risk and profit we aim to gain. In general, to earn more, larger risks must be considered. On the other hand, small hazards will imply smaller profits. In the general case, a portfolio can be composed of one or more of the following alternatives:

- Shares: indivisible capital unit expressing the ownership relationship between the company and the shareholder. Shares are registered titles that represent participation in the firm's equity. In the case of a company's default, the shareholders are the last ones to be refound. With the share, you become eligible to get a part of the dividend. The risk is high as the expected return.
- Bonds: debit certificate that describes the debitor's duties to the creditor. A firm/state contracts a debit that has to repay in a determined amount of time with some interest to the creditor. In general, a fixed amount of interest is returned at the end of every year. The bonds' logic is pure mathematics and it's linked to interest rates, inflation, duration and default probability. The risk is moderate and the returns are usually not so high.
- Commodities: Exchangeable raw material allowed via derivative tools (futures). In this case, the buyer sets the future buying or selling price of certain raw materials. For instace, the oil price is set to 50 dollars every tank from now to

one year. If the following year the cost of an oil tank is higher than 50 it's possible to make a profit otherwise it's a loss. The risk is very high and the return can be incredibly positive or negative.

- Cash: Having cash available allows to exploit new market opportunities. If only a part of the total amount of money is invested the risk of default is lower. Moreover, if the market offers new opportunities, it's possible to take them.

1.2 Asset

All the tools introduced by now are also called assets. An asset is an item with a certain economic value that can be bought, sold or rented. In this thesis, we consider as assets only the cash and the stocks. For this analysis, bonds are not considered for their intrinsic mathematics logic. In other words, to choose the best bond, we need only to compare the time required and the expected return. No machine learning methods are needed for this type of tool. Futures are not considered since it's difficult to model a problem that deals with this type of data. So, the portfolio is constituted of m different assets, where $m-1$ is the number of firms' stocks selected and the other one is the cash. The portfolio A_t at time t is the proportion of asset investment expressed in percentage as:

$$A_t = [a_t^0, a_t^1, ..., a_t^{m-1}] \tag{1.1}$$

where:

$$\sum_{i=0}^{M-1} A_t^i = 1 \tag{1.2}$$

In our formulation, we consider a_t^i always greater or equal to zero, but this is not the general case. Sometimes is it possible to find $a_t^i < 0$, namely, we are selling an asset that we do not own. This process is called short selling or shorting. The real mechanism is more difficult than that but the main stages are:

1. Borrow an asset (in general from a bank) and sell it at price p at time t;
2. Buy the asset again at a certain time $t+k$ at the new price p_{t+k};
3. Return the asset to the lender adding some money for the rent.

If $p_{t+k} < p_t$ the shorting is successful otherwise you can end up with a huge loss. While with the stocks the possible profit can be theoretically infinite and the

loss at most 100%, here we have the opposite situation. In shorting, if the stock price goes to zero, you can earn at most 100% otherwise, you can lose theoretically an infinite amount of money. For this parallelism, we prefer to avoid shorting in favour of long trading.

1.3 Prices

In this work, for every asset, on a certain day, four types of prices are taken into account: Open-High-Low-Close (OHLC). For instance, O_t^i stands for the open price of the i^{th} asset at discrete time t. Hence, the T-samples price time-series of an asset i, is the column vector:

$$o_{t:t+T}^i = \begin{bmatrix} o_t^i \\ o_{t+1}^i \\ . \\ . \\ o_{t+T}^i \end{bmatrix} \tag{1.3}$$

If we consider k assets for our analysis we can write:

$$\mathbf{O}_{t:T} = \begin{bmatrix} o_t^0 & o_t^1 & o_t^2 & \cdot\cdot & o_t^k \\ o_{t+1}^0 & o_{t+1}^1 & o_{t+1}^2 & \cdot\cdot & o_{t+1}^k \\ o_{t+2}^0 & o_{t+2}^1 & o_{t+2}^2 & \cdot\cdot & o_{t+2}^k \\ o_{t+T}^0 & o_{t+T}^1 & o_{t+T}^2 & \cdot\cdot & o_{t+T}^k \end{bmatrix} \tag{1.4}$$

Obviously, this definition can be extended to every type of price in OHLC. In general, absolute asset prices are not directly useful for an investor or for any type of AI or ML model. In Figure 1.1 we can see three-time series from 2000 to 2020 of vary famous brands: Apple Inc. (AAPL), 3M company (MMM) and JP Morgan Chase (JPM). For all three brands, the increasing trend is quite remarkable: JPM and 3M have doubled the stock value, while AAPL sees a growth of more than sixty times of its value. This fact is clear plotting the frequency distribution of the prices for the different assets. Dealing with data that is not normalized or stationary is not a trivial problem for ML algorithms. It's clear that for each asset trend, the mean, variance and autocorrelation structure change over time. Even normalizing the input data is not so automatic. If we normalize every stock independently, we will likely have different normalizations for every asset. This solution implies that normalized value 0.9 for "AAPL" corresponds to fifty dollars, while for "3M", the same normalized price corresponds to seventy dollars. This solution can fool the algorithm since we lose the different price magnitude between the prices.

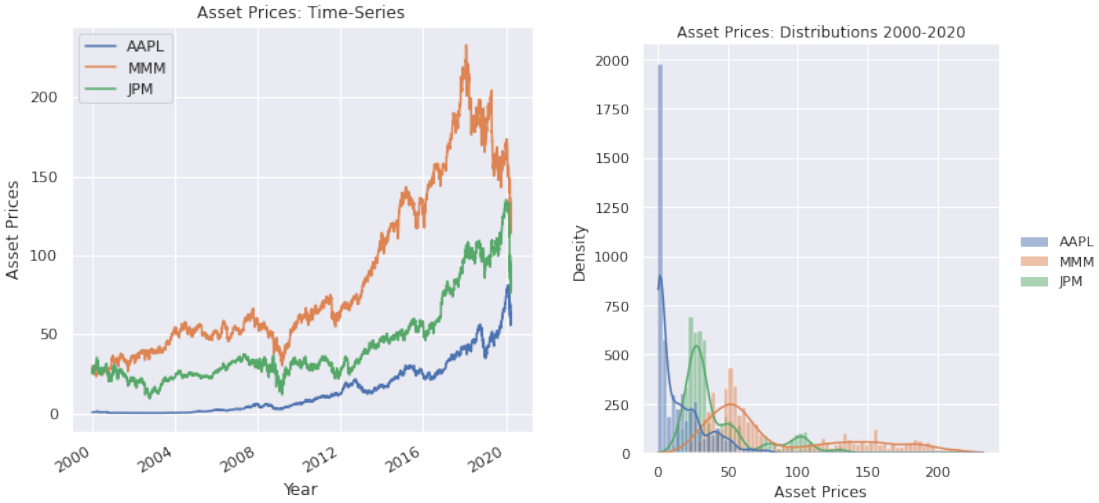


Figure 1.1: Time-series and distribution of the asset prices in the interval 2000-2020.

1.4 Returns

The algorithm as the investor can have problems making decisions based only on the absolute asset prices. In fact, what is crucial in this type of task is the changes of price over time since they can reflect the investment profit and loss, or more compactly, its return. In our analysis, we consider the changes over one day only. In this way, we can mitigate the noise present during a day of trading.

1.4.1 Gross returns

The gross return G_t of an asset represents the scaling factor of an investment in the asset at time $(t-1)$ [3]. For example, a B dollars investment in an asset at time $(t-1)$ will worth $B \cdot G_t$ dollars at time t. So, the gross return of asset i at time t can be computed as:

$$G_t^i = \frac{o_t^i}{o_{t-1}^i} \tag{1.5}$$

The asset gross returns are concentrated around unity and their behaviour does not vary over time for all stocks, making them attractive candidates for stationary autoregressive (AR) processes [4].

1.4.2 Simple returns

Very similar to the concept of gross returns are the so-called simple returns. The latter is more used since is more interpretable and represents the percentage change in asset price from time $(t-1)$ to time t. The computation is very similar to the one of the gross returns:

$$S_t = \frac{o_t - o_{t-1}}{o_{t-1}} = \frac{o_t}{o_{t-1}} - \frac{o_{t-1}}{o_{t-1}} = G_t - 1 \tag{1.6}$$

In other words, a simple return of $+0.1$ means that the asset value is increased by ten per cent to the previous day. Figure 1.2 clearly shows the simple returns of the aforementioned firms and their corresponding distributions. The plot on the left seems noisier with respect to the previous one but posses a lot of good properties. Now, we have zero-centred distributions with two relevant properties: stationarity and normalization. Therefore, we can use simple returns as a comparable metric for all assets thus enabling the evaluation of analytic relationships among them, despite originating from asset prices of different scales. Moreover, the distributions have a shape that approximate the normal one. Most of the machine learning algorithms works better with this type of distribution, starting from the shallow approaches to the deeps ones.

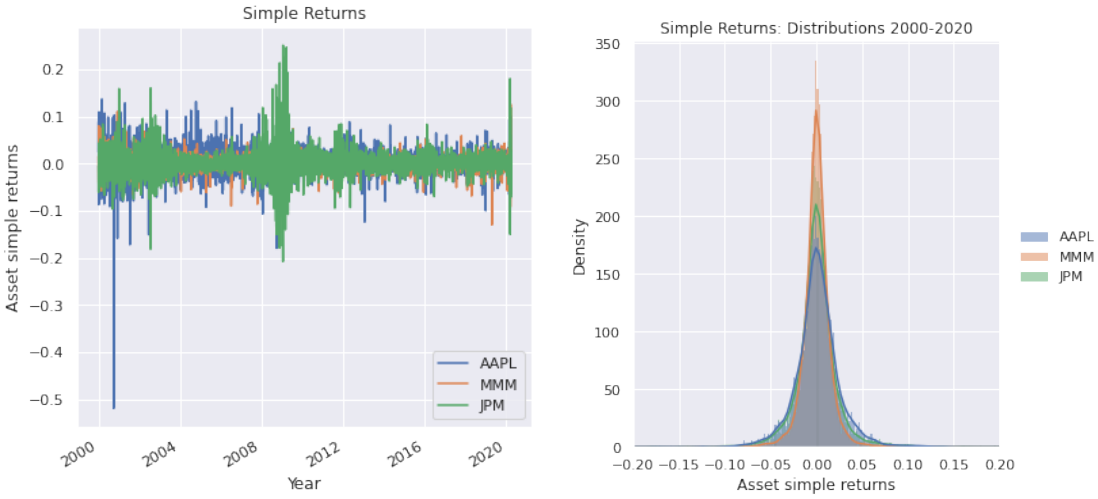


Figure 1.2: Time-series and distribution of the simple returns in the interval 2000-2020.

1.4.3 Log returns

The log-return at time t can defined as the logarithmic of the Gross returns:

$$L_t = ln(G_t) = ln(\frac{p_t}{p_{t-1}}) \tag{1.7}$$

The plot shown in Figure 1.3 is very similar to the one of the simple returns. This depends on the fact that the frequency distribution of the gross returns is centred around one but, thanks to the logarithmic operator, the log-returns are again centred around zero. Despite the interpretability of the simple return as the percentage change in asset price over one period, it is asymmetric and therefore practitioners tend to use log returns instead, to preserve interpretation and to yield a symmetric measure [4]. In other words, if the price of an asset initially increase by 15% and then decrease by 15% the final price is not equivalent to the first one.

$$100 \xrightarrow[Simple-Returns]{+15\%} 115 \xrightarrow[Simple-Returns]{-15\%} 97.75$$

On the other side, a 15% log increase in price followed by a 15% log decrease in price returns to the initial value of the asset.

$$100 \xrightarrow[Log-Returns]{+15\%} 115 \xrightarrow[Log-Returns]{-15\%} 100$$

This symmetry property added to stationarity and normalization of the frequency distribution makes this preprocessing strategy the most suitable for the problem of portfolio management and selection.

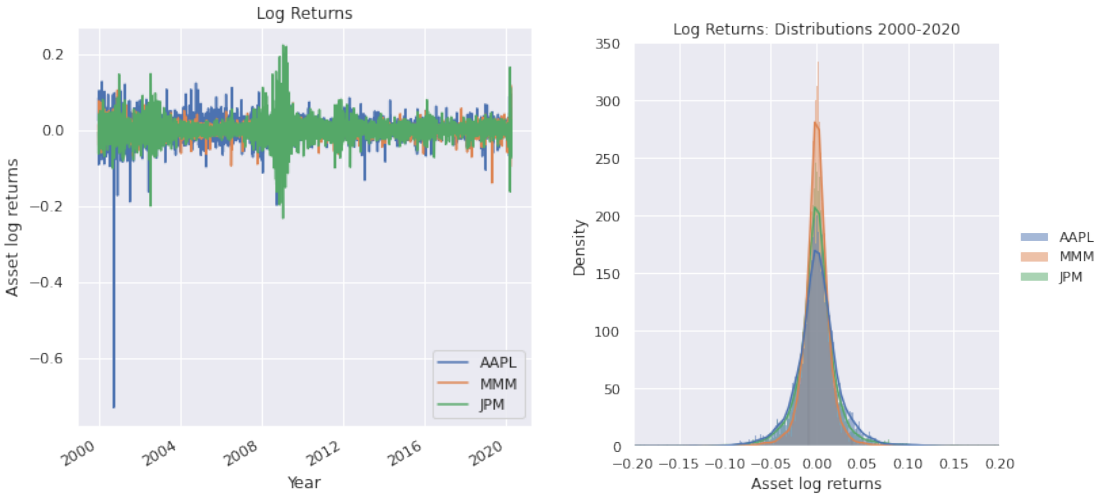


Figure 1.3: Time-series and distribution of the log returns in the interval 2000-2020.

1.5 Distributions comparison

At this point, let's understand both visually and theoretically why Log-returns are better than the usual time series. First of all, let's split our dataset into training and test set, as in any other ML problem. Let's consider the time between 2000 and 2018, the period in which the algorithm will be trained. While the time between 2019 and 2020 is the period where we are going to make predictions. Figures 1.4 and 1.5 show major differences in the density distribution between training and test set. In the case of prices, we have major changes that can really adversely affect the performance of the ML algorithm. To have a reliable result, the distribution of the training and test set must be similar and obviously, this is not the case. The standard deviation is higher, but it's quite usual since in the test split we have fewer samples, but the mean value is very different for all three assets. This result was expected since the asset prices are not stationary nor normalized. The Log-returns, thanks to all the properties presented in the previous chapter, overcomes the distribution problems of the asset prices. Now, both in the training and the test set we have a normal-shape distribution centred around zero.

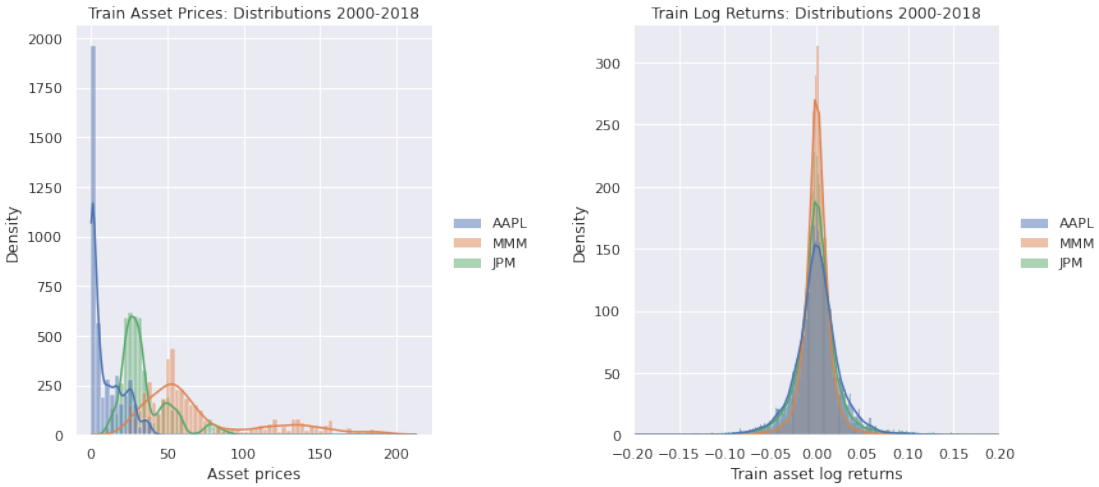


Figure 1.4: Distribution of the prices and the log returns in the training period 2000-2018.

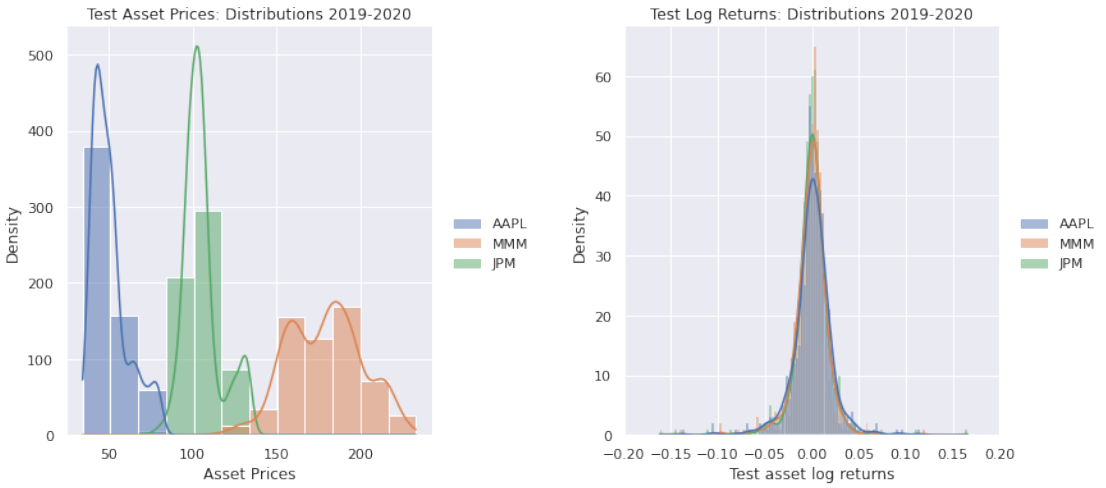


Figure 1.5: Distribution of the prices and the log returns in the testing period 2019-2020.

1.6 Risk-Ratios

At this point, let's introduce formally what Risk is. Risk is defined in financial terms as the chance that an outcome or investment's actual gains will differ from an expected outcome or return. Risk includes the possibility of losing some or all of an original investment. Risk is usually assessed by considering historical behaviors and outcomes. In finance, standard deviation is a common metric associated with risk. Standard deviation provides a measure of the volatility of asset prices in comparison to their historical averages in a given time frame [5]. If we consider different investments with the same expected return, a good strategy is to choose the one with the lowest risk. If we consider strategies with distinct returns, further considerations and aspects have to be taken into account, especially if the investment with the highest return is not the one with the lowest risk. Starting from the idea of risk and returns, it is possible to study a lot of indexes that indicate the performances of our portfolio investments. Mainly in our analysis, we will consider:

1. The Sharpe Ratio (SR) was ideated by the economist and Nobel price winner William Sharpe. The main idea behind this index is to weigh the returns by the risk taken. So, if an investment has high returns and risks the strategy will not be considered so good. The SR values are in the interval $[-\infty, 1]$. Generally, investors consider an investment "profitable and safe" if the SR overcomes 1. Obviously, this threshold can change by looking at the portfolio investments and results of our competitors. To calculate the Sharpe Ratio,

investors first subtract the risk-free rate from the portfolio's rate of return, often using U.S. Treasury bond yields as a proxy for the risk-free rate of return. Then, they divide the result by the standard deviation of the portfolio's excess return. The exact formula for calculating the Sharpe Ratio is as follows:

$$SR = \frac{R_p - R_f}{\sigma_p} \tag{1.8}$$

where R_p is return of our portfolio, R_f is the risk free return and σ_p is the risk of our portfolio [6].

2. The Sortino Ratio (SoR) is a simple variation of the SR that exploit the downside risk over the usual risk. The idea is to consider only the negative returns which are the ones that effectively worries the investors. Computing the risk in this way, we obtain the so-called downside risk. The computation to obtain this index is similar to the previous one:

$$SoR = \frac{R_p - R_f}{\sigma_{dp}} \tag{1.9}$$

where R_p is return of our portfolio, R_f is the risk free return and σ_{dp} is the downside risk.

Chapter 2

Reinforcement Learning

2.1 Markov processes

2.1.1 Introduction

This chapter is mainly inspired by the DeepMind [7] [8] and the Berkley [9] reinforcement learning (RL) courses and the book of Abhijit Gosavi [10]. In this chapter, we introduce various RL algorithms that could be useful in the Portfolio management and selection field. We analyse each of them and we will select the ones that seem more suitable for our task.

2.1.2 Definitions

Before talking about Markov Process (MP), Markov reward Process (MRV) and Markov decision Process(MDP), we need to introduce the following concept and notations:

- The **decision-maker**, also called the agent, is the entity that selects the control mechanism.
- **Action space** (A) is the set of all possible actions the decision maker can take in the environment.
- The **state space** (S) is the set of all possible states. A state at time t (s_t), is a representation of the environment. Formally, the state is a function of history. A state, to be Markovian, must satisfy the following property:

$$P[s_t|s_0, s_1, ..., s_{t-1}] = P[s_t|s_{t-1}] \tag{2.1}$$

  This formula implies that future evolution is independent from the past, given the present state. Namely, we assume that the stochastic process x is

homogenous. This means that the state at every time step t embeds all the information of the underlying system.

A

- **Stochastic process** is defined as a function $x : T\, x\, \Omega \rightarrow S$. Where $T = [0, +\infty]$ is the parameter set and S is the state space.
- A **reward** at time t $(R_t) : (S, A) \rightarrow \mathcal{R}$, at time t is a scalar signal that provides feedback to the agent on its performance at step t. The agent aims to maximise the cumulative reward.
- The **transition probability matrix** (TMP) $(\rho_{ss'} = P(x_{t+1} = s'|x_t = s))$ gives the one step probability, starting from state s, to end up in state s'. In an MDP, we usually have a different TMP associated with each action a. We indicate it as $\rho^a_{ss'}$.
- Finally, we have the concept of policy. The **policy** is a function $\pi : S \rightarrow A$ that describes agent's behaviour. In general, we can distinguish two different kinds of policy:
  1. A *stochastic* one, $\pi(a|s) = P[A_t = a|S_t = s]$. In this case, for each state s, we have a different probability distribution from which we can sample the new action a'.
  2. A *deterministic* one, $a = \pi(s)$. In this case, given a state s, the agent knows deterministically the action to follow. In other words, given the same state more than one time, the agent will perform the same action again, again and again.

2.1.3 Markov process

A Markov process (or chain) is a memory-less random process defined by the tuple $< S, \rho >$. In this environment all states are Markovian. This process was ideated by the russian mathematician Andrej Andreevič Markov and nowadays is used in many fields like telecommunications and queuing theory. A clear example of that is shown in Figure 2.1.

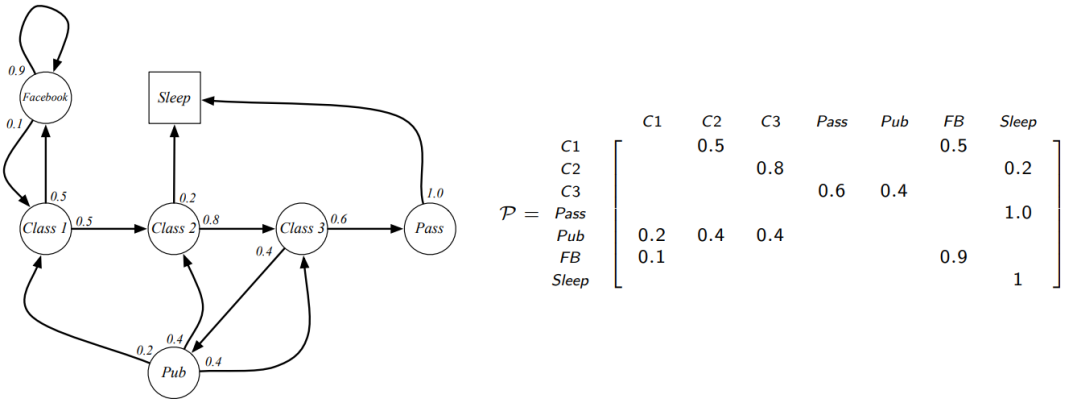


Figure 2.1: Markov Process example with relative transition rate matrix [8].

In this case, every circle represent a different state from which we can move to a new one. On the other hand, the squares denote terminal states: as the name said, once we reach one of them, we can not come back. If we see this representation as a directed graph, the square states can be seen as trapping nodes. Based on a corollary of the "Finiteness of expected hitting times"[11] theorem, if we consider a random walk $X(t)$ on this directed graph with trapping set of nodes V then, with probability 1, there exists a time $\bar{t}$ such that:

$$X(t) \in V \quad \forall t > \bar{t} \tag{2.2}$$

2.1.4 Markov reward process

A Markov Reward Process is a tuple $< S, \rho, R, \gamma >$. Unlike the MP, in this new random process, we consider a reward function R and a discount factor $\gamma \in [0, 1]$. R_t will represent the reward obtained by the agent at time t. At this point, we can introduce the concept of Return (G_t) which represents the total discounted reward from time-step t, defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_k \gamma^k R_{t+k+1} \tag{2.3}$$

Discount γ is the present value of future rewards. The idea of discounting is related to the fact that the value of money reduces with time. So, for instance, if a person earns 3 euro today, 5 euro tomorrow, 6 euro the day after tomorrow, and if the discounting factor is 0.9 per day, then the present worth of this person's earnings will be:

$$G_t = 3 + 0.9 \cdot 5 + 0.9^2 \cdot 6 \tag{2.4}$$

The reason for raising 0.9 to the power of 2 is that tomorrow, the present worth of day-after-tomorrow's earning will be 0.9(6). Hence today, the present worth of this amount will be $0.9 \cdot [0.9 \cdot (6)] = (0.9)^2 \cdot 6$ [10].

At this point, with all these concepts, we can try to understand how good is to stay in a state s compared to state s'. To do this, we can use the value function $v(s)$ which gives the long-term value of state s. The state value function $v(s)$ of an MRP can be defined as the expected return from state s:

$$v(s) = E[G_t|S_t = s] \tag{2.5}$$

As we have previously seen, the process is homogenous so, the function v is independent from the time t. Starting from the definition of G_t we can rewrite:

$$v(s) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \tag{2.6}$$

$$v(s) = E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...)|S_t = s] \tag{2.7}$$

The quantity in the square bracket can be rewritten in term of return (G) so:

$$v(s) = E[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{2.8}$$

$$v(s) = E[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \tag{2.9}$$

This formula is also known as the Bellman Equation. If we compute the expected value we finally get:

$$v(s) = R_s + \gamma \sum_{s'} \rho_{ss'} v(s') \tag{2.10}$$

Solving this system of equations has a computational cost of $O(n^3)$, where n represents the number of states. Hence, a straightforward solution is possible only when we have small MRPs. For this reason there are many iterative methods for large MRPs, e.g.

1. Dynamic programming
2. Monte-Carlo evaluation
3. Temporal-Difference learning

2.1.5 Markov decision process

A Markov decision process (MDP) is a Markov reward process with decisions. In this case, the random process is a tuple $< S, A, \rho, R, \gamma >$, where A is a finite set of actions. The state-value function $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π:

$$v_\pi(s) = \sum_a \pi(a|s) q(s,a) \tag{2.11}$$

where, the action-value function, $q_\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π:

$$q_\pi(s,a) = E_\pi[G_t | S_t = s, A_t = a] \tag{2.12}$$

As already seen for the value function v, the function q does not depend on the time t. This new function can also be written as:

$$q_\pi(s,a) = R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_\pi(s') \tag{2.13}$$

Hence, $v_\pi(s)$ is a kind of weighted sum of the returns from state s times the probability of taking action a. At this point, we can substitute the value of 2.13 in the definition of 2.11:

$$v_\pi(s) = \sum_a \pi(a|s) \left(R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_\pi(s') \right) \tag{2.14}$$

The latter formula is also known as the Bellman expectation equation. This allow us to evaluate the value function given a certain policy π but, the goal is to find the optimal value function that specifies the best possible performance for the MDP. An MDP is "solved" when we know the optimal value function. Hence, the optimal state-value function $v_*(s)$ is the maximum value function on all policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.15}$$

Similarly, the optimal action-value function $q_*(s,a)$ is the maximum action-value function on all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.16}$$

But what does max mean for a policy? $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for each state s. In particular, for MDP there is always a policy π_* better or equal to all other policies $\pi_* \geq \pi \ \forall \pi$. At this point, to find the optimal policy, it's possible to maximize $q_*(s,a)$. Hence, $\pi(a|s) = 1$ if $a = \arg\max_a q_*(s,a)$ [7] , 0 otherwise. This

means that there is always a deterministic optimal policy for any MDP. Starting from that we can obtain the Bellman optimality equation:

$$q_*(s,a) = R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_*(s') \tag{2.17}$$

and replace it in the formula of $v_*(s)$:

$$v_*(s) = \max_a R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_*(s') \tag{2.18}$$

2.2 Dynamic Programming

2.2.1 Introduction

Since we are dealing with a system of linear equation, where the number of unknown variables (all independent) coincides with the number of equation, both the Bellman equation 2.10 and the Bellman Expectation equation 2.13 can be solved in a closed-form. To solve this system of equations, the number of computations required is: $O(n^3)$ where n is the number of states. This means that, even with a small problem of 1000 states, we have a computational effort of one billion steps. Moreover, if we consider the Bellman optimality equation, due to non-linearity, we do not have a closed-form solution. At this point, we can introduce a method for solving complex problems called *dynamic programming* (DP). This approach works in the following way:

1. split the problem into sub-problems
2. find the optimal solution for each sub-problems
3. combine the intermediate results to form the optimal solution to the starting problem.

To apply DP, problems must satisfy the optimal substructure property namely, the optimal solution can be decomposed into sub-problems. Since we are dealing with MP problems, we know that the future evolution of the system is independent from the past, given the present state. This implies that DP can be applied to MDP. Moreover, DP is very effective when dealing with sub-problems that recur many times and, for this reason, the solutions can be cached and reused many times. These requirements fit perfectly with MDPs since the Bellman equation gives recursive decomposition and the value function stores and reuses solutions.

2.2.2 Iterative policy iteration

Dynamic programming assumes full knowledge of the MDP and can be used both for prediction and control. In particular, with the Iterative Policy Evaluation (IPE), we want to evaluate a given policy π. The solution is an iterative application of the Bellman expectation equation using synchronous backup as:

$$v_\pi(s) = \sum_a \pi(a|s) \left(R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_{pi}(s') \right) \tag{2.19}$$

The IPE algorithm works in the following way:

1. Initialize $k = 0$ and $v^0 = \mathbf{0}$ $\forall s \in S$ set ϵ to a small value like 10^{-2} or 10^{-3}
2. at each iteration k+1 and for all states $s \in S$ update $v^{k+1}(s)$ from $v^k(s')$ using:
$$\mathbf{v}^{k+1} = \mathbf{R}^\pi + \gamma \mathbf{P}^\pi \mathbf{v}^\mathbf{k} \tag{2.20}$$
where s' is a successor state of s.
3. if $||\mathbf{v}^{k+1} - \mathbf{v}^k|| > \epsilon$ come back to step 1 and increase k by one unit. Otherwise stop, the solution is $\mathbf{v}^{k+1}$.

A clear example of IPE is the one applied to the "Small Gridworld", as shown in Figure 2.2 . So, we consider an undiscounted MDP ($\gamma = 1$) with 16 states. Two of them are terminal, represented with a grey square. We obtain a reward of -1 until one of the two terminal states are reached. The agent, in this case, follows a random policy: this means that he can move in any direction with a probability of $\frac{1}{4}$. If the agent tries to "escape" from the grid, the state remains unchanged. At $k = 0$, we start with a value function full of zeros. Only after three steps it is quite clear what a better policy could be. And this new policy does not change even with $k \to +\infty$.

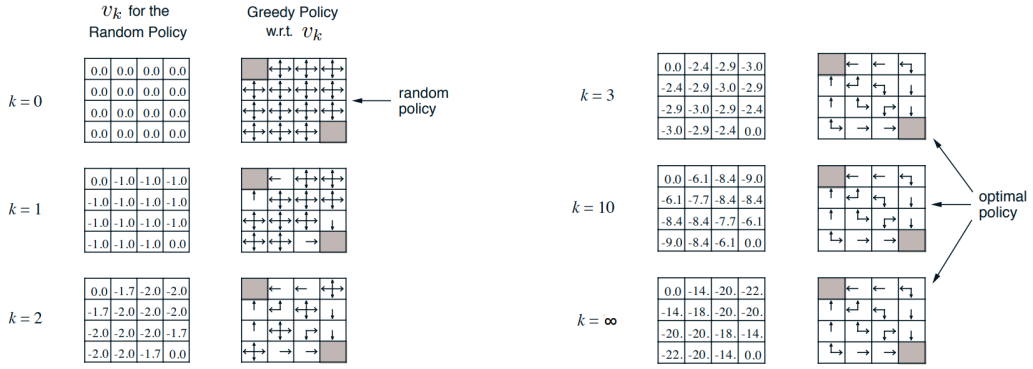


Figure 2.2: Iterative Policy Evaluation in Small Gridworld [8].

2.2.3 Policy Iteration

At this point, we want to improve a given policy π iteratively. So, the first idea is to evaluate the policy and then improve it by acting greedily on v_π. But what acting greedily means? Acting greedily means to find, for every state s, a deterministic policy that select the best action that maximizes the action-value function.

$$\pi' = \arg\max_{a \in A} \quad q_\pi(s, a) \tag{2.21}$$

At this point, we have that:

$$q_\pi(s, \pi'(s)) = \max_{a \in A} \; q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s) \tag{2.22}$$

This expression means that if we follow policy π but in the state s we compute action $\pi'(s)$ we have, by definition of $\pi'(s)$, that the action-value function $q_\pi(s, \pi'(s))$ is greater or equal to the action-value function obtained following policy π at every step. We iterate this process, as shown in Figure 2.3 until no further improvements are possible. Then the Bellman optimality equation has been satisfied:

$$v_\pi(s) = \max_{a \in A} q_\pi(s, a) \tag{2.23}$$

where $v_\pi(s) = v_*(s)$ for $\forall S$. Hence, when the algorithm converges, we are sure to have found the best policy. But, as we have seen in the "Small Gridworld" environment, only after three steps the best policy could be selected. So, a possible modification to this algorithm could be to change the stopping criteria. For instance, we could stop when the difference between the new value function and the old one is below a certain threshold ϵ. Another possibility, as seen before, could be to stop the algorithm after k iterations.

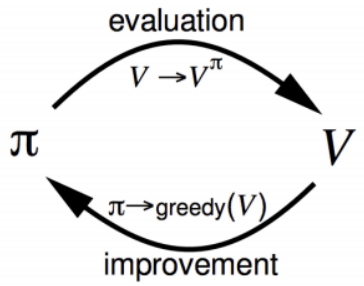


Figure 2.3: Policy loop [8].

2.2.4 Deterministic Value Iteration

In the Value Iteration (VI) approach, we do not use the Bellman expectation equation anymore but, we start from the idea that we know the solution of our

subproblems $v_*(s')$. Then the solution $v_*(s)$ can be found by one-step lookahead using the Bellman optimality equation:

$$v_*(s) = \max_a R_t^a + \gamma \sum_{s'} \rho_{ss'}^a v_*(s') \tag{2.24}$$

The idea of value iteration is that we start with some arbitrary value of $v_*(s')$ and iteratively update the value of $v_*(s)$. This method will converge to the optimal value function. The intuition behind VI is that we start with the final reward and, then, we propagate this reward backwards and we update step by step the value function of each state. A clear example is the one of the shortest path 2.4. The states next to the goal at the first step get the reward of -1, since they need a step to achieve the goal. In the following step, the states next to the red one get a reward of -2 since they need two steps to achieve the goal. As we said before, the final reward starts to propagate backwards like a (colour) wave until it arrives at the farthest state (the pink one).

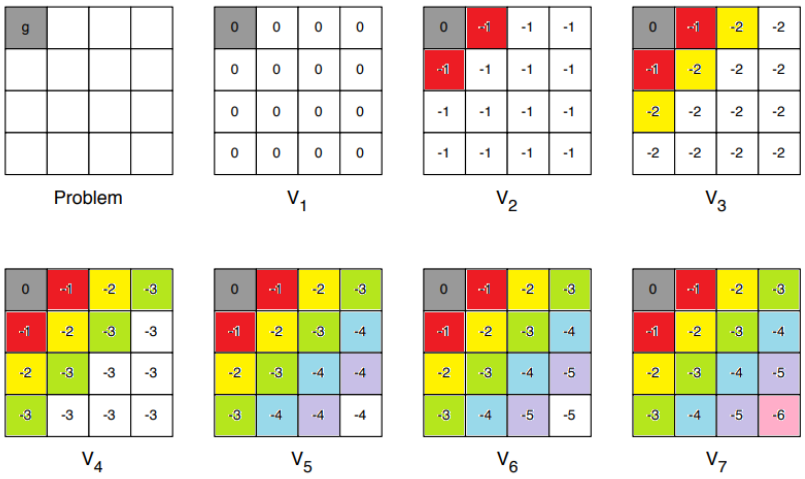


Figure 2.4: Shortest path value function update.

2.3 Model free prediction and control

2.3.1 Introduction

Until now, we have supposed to know the MDP that we were analysing. This means that we know the full dynamic and the reward function of that MDP, then using Dynamic Programming (DP) and Bellman equations, we were able to determine the best policy π_*. The fact to know the MDP is a quite strong assumption and

makes Reinforcement Learning (RL) useless for most interesting problems and applications. For this reason, we introduce Model Free methods that can be used for prediction and control problems without knowing the dynamics of the MDP. In this case, we try to approximate the value function based only on the experience of the agent. Before moving forward, we have to introduce the concepts of bootstrapping and sampling.

- **Bootstrapping** means that when we update our prediction of the value or the action-value function, we do that based on an estimated value of the next step.
- **Sampling** means that we do not consider all possible actions that the agent can choose in a certain state, but we sample from that, picking a single action.

2.3.2 Monte-Carlo Learning

Monte-Carlo (MC) learning is a model-free method with no knowledge of the transition probability matrix $\rho_{ss'}$ and the reward function R_s. For this reason, this method learns directly from episodes of experience. This solution has both some advantages and some drawbacks. The main pro is that using the true value of the total discounted reward, G_t MC is an unbiased estimator of the value function $v(s)$. Another important aspect is that using sampling this algorithm can be many times more efficient than DP. The major drawback is that this method does not use bootstrap; this implies that to use this algorithm, we need complete episodes. This fact implies another crucial problem, namely, that this method can be used only in an offline environment. In fact, in an online setting, we can not immediately know the final value of G_t. So, in an MC policy evaluation problem, the goal is to learn v_π from episodes of experience under policy π. Hence, to evaluate a state s the first (or even every time is practically the same) time t that the state is visited in an episode:

- Increment a counter $N(s) \leftarrow N(s) + 1$
- Increment the total return $T(s) = T(s) + G_t$
- $v(s)$ is finally estimated by averaging T(s) and N(s): $v(s) = \frac{T(s)}{N(s)}$

The estimated $v(s)$ will tend to optimal $v_*(s)$ thanks to the law of large numbers. At this point, we can reformulate the solution in an equivalent way that will be useful in the following topic. In fact, knowing the value G_t, we can update the value function $v(s)$ step by step, and we do not have to wait until the end of the episode:

$$N(s_t) \leftarrow N(s_t) + 1 \tag{2.25}$$

$$v(s_t) \leftarrow v(s_t) - \frac{1}{N(s_t)}(G_t - v(s_t)) \tag{2.26}$$

In practice, we adjust the value of $v(s_t)$ in the direction of G_t. While for a non-stationary problem the true value is going to vary over time so, intuitively, an approach that converges to a single value is not going to work well. Hence, instead of weighting for the reciprocal of the counter we use a value α that acts like a learning rate.

$$v(s_t) \leftarrow v(s_t) - \alpha(G_t - v(s_t)) \tag{2.27}$$

2.3.3 Temporal-Difference Learning

Temporal-Difference (TD) method is very similar to the Monte-Carlo one but differs from the previous one for exploiting the sampling property. In this case, we update the value of $v(s)$ using an estimated return instead of the true value. The new algorithm becomes:

$$v(s_t) \leftarrow v(s_t) - \alpha(R_{t+1} + \gamma v(s_{t+1}) - v(s_t)) \tag{2.28}$$

Where $R_{t+1} + \gamma v(s_{t+1})$ is called the TD target and $R_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is called the TD error. Using an estimated value of the return, we end up with an estimator that is not unbiased anymore. At a first look, this could be a negative side of this algorithm but, in practice, this approach can reduce the variance and positively affect the final result. In fact, while in MC the return depends on many actions, transitions and rewards, in TD, the return depends only on a single action, transaction and reward. Moreover, using TD, we can overcome some crucial problems of the Monte-Carlo approach. First of all, TD can learn from incomplete sequences, while MC can only work with complete ones. Then, this implies that TD can work in continuing and non-terminating environments. At this point, a possible idea to improve this algorithm is to look not only one step ahead but n-steps ahead. Practically we will change the TD target from:

$$TD - target^{(1)} = R_{t+1} + \gamma v(s_{t+1}) \tag{2.29}$$

to:

$$TD - target^{(n)} = R_{t+1} + ... + \gamma^{n-1} v(s_{t+n}) \tag{2.30}$$

This means that the MC algorithm is a special case of the TD when n coincides with the episode's length. Another possible improvement could be average these n-step returns over different n. For instance, we could average the one-step with the two-step return:

$$\frac{1}{2}G^{(1)} + \frac{1}{2}G^{(2)} \tag{2.31}$$

In this way, using this mean, we obtain a new td-target that is more robust with respect to the previous one. So, to combine all the n without increasing the algorithm complexity we introduce the so-call λ-return that works as a geometric weight sum over all n. The new update formula based on this idea is called forward-view TD(λ) and is defined as:

$$v(s_t) \leftarrow v(s_t) - \alpha(G_t^\lambda - v(s_t)) \tag{2.32}$$

where:

$$G_t^\lambda = (1-\lambda)\sum_n \lambda^{n+1} G_t^{(n)} \tag{2.33}$$

where (1 - λ) is a normalizing factor to sum up to one. Using this formula, we have to look forward to compute G_t^λ , hence, we can deal only with complete episodes like in MC. Luckily we can overcome this problem with the concept of **Eligibility trace**. In practice, for each state, we trace how frequent we visit that state and how recently we have visited that state. Eligibility trace for state s can be defined:

$$E_0(s) = 0 \tag{2.34}$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s) \tag{2.35}$$

At every time step t, we discount the old value of E and we increase it by one unit if the state s is visited. Combining this with $TD(\lambda)$ we obtain the Backward View TD(λ).

$$v(s) \leftarrow v(s) + \alpha\delta_t E(s) \tag{2.36}$$

where:

$$\delta_t = R_{t+1} + v(s_{t+1}) - v(s_t) \tag{2.37}$$

namely δ_t represents the td-error. If we set the value of $\lambda = 0$, we solely come back to the initial implementation of TD.

2.3.4 Introduction to control

Model-free control aims to optimise the value function of an unknown MDP. Different methods can be used for solving the control problem and, the main discriminatory element among them is how they learn. We have two different types of learning:

- On-policy: Learn about policy π from experience sampled from π

- Off-policy: Learn about policy π from experience sampled from μ. Practically they learn by observing other agents' behave or re-using experience generated from old policies.

That said, to be able to select the best policy, the model should exploit all the possible state-action tuples. Using greedy action selection, the risk is to not be able to explore all them. This fact could imply that we can select the right policy that maximise the final reward. Hence, the simplest idea for ensuring continual exploration is to fix a parameter ϵ and then:

- with probability $1 - \epsilon$ selects the greedy action
- with probability ϵ selects an arbitrary action

Moreover, it can be demonstrated that if $\epsilon \to 0$ for $number - of - episodes \to +\infty$ then the ϵ-greedy policy $\pi \to \pi_*$. This particular condition is also called GLIE (Greedy in the Limit with Infinite Exploration).

2.3.5 Sarsa

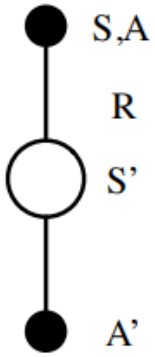


Figure 2.5: Updating Action-Value function with SARSA [8].

Sarsa in an on-policy method that apply TD learning to update the action-value function $Q(S, A)$. The reason why this algorithm takes this particle name is immediately clear if we look at Figure 2.5. This picture is quite self-explicative on how this method works. We start in a certain, state-action pair (S, A), and, taking action A, we get a reward R and we end up in a new state S'. At this point, we select a new action A' ϵ-greedy based on our actual policy π. Finally, we can update our action-value function $Q(S, A)$ based on the value of $Q(S', A')$.

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)) \tag{2.38}$$

This means that we do not use the exact return G_t but, we bootstrap and finally sample a new action A'. Then we update the current action-value function $Q(S, A)$ in the direction of the td-error. Sarsa converges to the optimal action-value function $Q_*(S, A)$ if:

- the GLIE conditions are satisfied
- the learning rate α respects the Robbins-Monro sequence, namely:

$$\sum_{k}^{+\infty} \alpha_k \to +\infty \tag{2.39}$$

$$\sum_{k}^{+\infty} \alpha_k^2 < +\infty \tag{2.40}$$

2.3.6 Backward View Sarsa

Finally, as already seen for TD learning, we can use eligibility traces for each state-action pair:

$$E_0(S, A) = 0 \tag{2.41}$$

$$E_t(S, A) = \gamma E_{t-1}(S, A) + \mathbf{1}(S_t = S) \tag{2.42}$$

The value γ determines how quickly new information should propagate backwards to the trajectory. This means that using this eligibility trace we can update our action-value function more quickly than the one-step Sarsa. Finally, the value of $Q(S, A)$ is updated by multiplying the td-error for the eligibility trace $E_t(S, A)$. We have to highlight that this new function E is re-initialized to zero at the end of each episode.

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))E(S, A) \tag{2.43}$$

2.3.7 Q-Learning

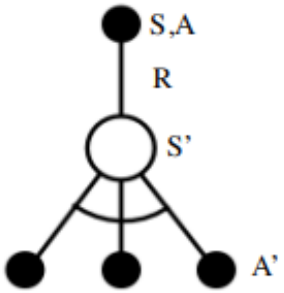


Figure 2.6: Updating Action-Value function with Q-learning [8].

The Q-learning method is an off-policy learning method that exploits the Bellman optimality equation to update the action-value function $Q(S, A)$. It's based on two different policy: a behaviour (μ) and a target (π) one. Starting in state S, the new action A is chosen ϵ-greedy following the behaviour policy. Taking action A, we end up in a new state S'. At this moment, the target policy comes into action. This means that the new action A' is chosen greedily with respect to Q(S, A):

$$\pi(S') = argmax_a Q(S', a) \tag{2.44}$$

This implies that the new Q-learning target becomes:

$$R_t + \gamma \, max_a(Q(S', a)) \tag{2.45}$$

while the Q-learning error can be rewritten as:

$$R_t + \gamma \, max_a(Q(S', a)) - Q(S, A) \tag{2.46}$$

Having said that the final Q-learning update rule:

$$Q(S, A) = Q(S, A) + \alpha(R_t + \gamma \, max_a(Q(S', a)) - Q(S, A)) \tag{2.47}$$

All this formule can be summed up in Figure 2.6. As Sarsa, the Q-learning control method converges to the optimal action-value function $Q_*(S, A)$.

2.4 Model-based algorithms

2.4.1 Introduction

What we have done so far is:

1. Acting following a specific policy π
2. Learn the feedback from the environment for that particular action
3. Then update our value-function/policy based on the new state and the corresponding reward
4. Repeat the process from 1.

These solutions work without directly knowing the MDP model. Hence, for this reason, they are called Model-free methods. This idea works without any type of assumption, and all algorithms converge to the optimal policy π. Hence, if the agent hadn't explored the entire state space, this optimal result can not be reached. So, the idea is to create an approximated MDP model to be able to run as much simulation as we want without the necessity of knowing the original model. A model-based works quite differently:

1. Acting following a specific policy π
2. Learn the feedback from the environment for that particular action
3. Improve our model based on the new experience
4. Using the model to plan and update value-function/policy
5. Repeat the process from 1.

With this solution, we can efficiently learn the model by supervised learning methods and reason about model uncertainty. On the other hand, the major drawback is that we have two different sources of approximation error: we construct an approximated value function using an approximated model. It's clear that if we try to learn from a wrong approximated model, we end up with a wrong answer.

2.4.2 Define a model

A model m is a representation of an MDP $< S, A, \rho, R >$, parametrized by η. If we assume that the state and the action space are known, we can rewrite our model $m =< \rho_\eta, R_\eta >$. Where ρ_η represents the transition rate probability function and R_η the reward function. Being in a state s_t and taking an action a_t, we will sample the next state s_{t+1} and the reward r_{t+1} respectivelly from ρ_η and R_η. To estimate ρ_η and R_η, we use the experience of the agent. Learning the reward function is a classical regression problem while learning the transition rate probability function is a density estimation problem. Even in this case, we have a regression problem that can be solved by picking a loss function (e.g. MAE, MSE, KL, HUBER, ...) and minimise it with respect to η. Examples of most used models are:

- Table Lookup that can be parametric or not parametric
- Linear Expectation model
- Linear Gaussian model
- Gaussian process model

2.4.3 Sample-Based Planning

At this point, we can train the model based on the experience of the agent. Once we obtained $m =< \rho_\eta, R_\eta >$. we can use the model to generate as much sample as we want. In this way, even if the agent's experience is limited, we can obtain a higher number of samples from our approximated model. Then, we can apply any model-free RL (e.g. Sarsa, Q-learning, ...) to the new synthetic samples. Often Sample-based planning methods are efficient than the standard ones. Even if could seem counter-intuitive, the sampled experience could help the method to exploit even more than before. A clear example comes from Figure 2.7. In the sample experience, we have the episode (A, 0, B, 1) that is not present in the real experience.

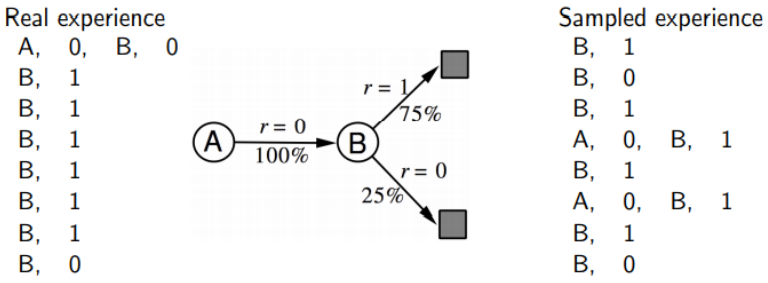


Figure 2.7: Real experience vs Sample experience [8].

2.4.4 Dyna

The main idea under Dyna is to combine the benefits of both model-free RL and model-based RL. As clearly shown in Figure 2.8, Dyna learns a model from real experience. Then it learns and plans a value-function from real and simulated experience. In this way, the method benefits from both kind of experiences and increase its performance remarkably. What happens is the following:

1. Starting from a state S, we select a new action A ϵ-greedy
2. Execute action A and observe the results

3. Based on the state and the reward, update the action-value function and the model

4. Then, for n (hyperparameter prespecified by the user) times, we update out action-value function again, based on the simulated experience.

5. Repeat the process from 1.

The model learns very quickly and can adapt itself also to a changing environment.

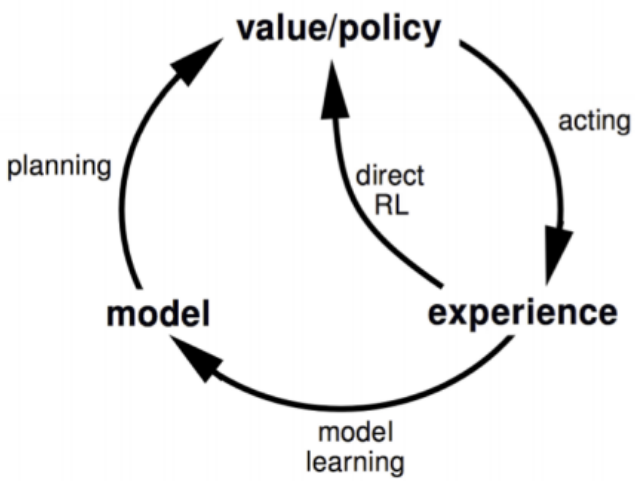


Figure 2.8: Dyna architecture [8].

2.4.5 Simulation-Based Search

Since now we have an approximated representation of the model, we can explore for each state s what happens if we choose different actions. Practically, we are building a search tree (Figure 2.9) with the current state s_t as the root node. At this point, we could use dynamic programming to solve the whole MDP but, this can be computationally expensive. As we have already seen, we do not need to solve the entire MDP, hist sub-MDP starting from the current state s_t. In fact, we can use any model-free RL methods that use sampling.

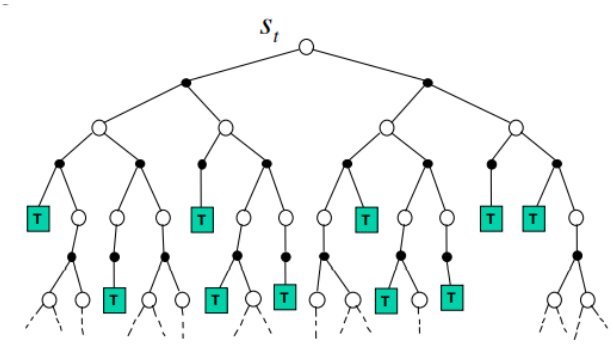


Figure 2.9: Forward search [8].

Monte-Carlo Tree search

Given a model, we can simulate K episodes from the current state using a certain policy π. For every state, we select a new action ϵ-greedy and, if we face a state not visited yet, we pick actions randomly. With these episodes, we can build a search tree containing visited states and actions. Then, for every possible state and action, we can compute the corresponding action-value function $Q(s, a)$ by the mean return of episodes from s, a. After the search is finished, we update the policy greedily.

$$a_t = argmax_a Q(s_t, a) \tag{2.48}$$

This is simply Monte-Carlo control applied to simulated experience. Hence, based on the MC control theory we already know that this method will converge on the optimal search tree, $Q(s, a) \rightarrow q_*(s, a)$. MC tree search thanks to sampling and the simulation approach is able to overcome the performance of DP and break the curse of dimensionality. Moreover, thanks to this algorithm structure MC is easily parallelisable and computationally efficient.

2.5 Exploration vs Exploitation

2.5.1 Introduction

One of the most crucial parts of RL algorithms is to find the correct trade-off between: Exploitation; take the best action according to the policy/value function. In this case, we try to maximise the total reward based on what we already know about the system response. Exploration; gather more information to understand if different paths can be followed to maximise the total reward. The level of exploration is highly dependent on the context. Maybe in an industrial factory, we prefer to not explore the entire state space to avoid accidents. While in a game

playing an experimental move can increase our knowledge of the game without any kind of safety risk. At this point, if we rewrite the action-value as the mean reward for action a:

$$Q(a) = E[r|a] \tag{2.49}$$

and the optimal value V^* as:

$$V^* = Q(a^*) = max_a Q(a) \tag{2.50}$$

Now, we want to introduce a new measure that establishes how much remorse we have if choose a random action a instead of the best action a^*. We can call this new dimension regret:

$$l_t = E[V^* - Q(a_t)] \tag{2.51}$$

Finally, if we sum all regret for every time-step until time t, we obtain the total regret:

$$L_t = E[\sum_{\tau}^{t} V^* - Q(a_\tau)] \tag{2.52}$$

Minimise the total regret is equivalent to maximise the cumulative reward. The problem is that we do not know the best action a^* (otherwise, we already know the answer to that problem) so, what we can do is to find a good trade-off between exploration and exploitation. This because if an algorithm forever/never explores, it will have a linear total regret. To try to achieve a sub-linear regret, we can follow different strategies:

- Naive Exploration: add noise to greedy policy
- Optimistic Initialisation: assume the best until proven otherwise
- Optimism in the face of uncertainty: prefer action with uncertain values
- Probability Matching: select actions according to probability they are best

2.5.2 Naive Exploration

Greedy algorithm

If we consider an estimate $\widehat{Q_t}(a) \approx Q(a)$, the greedy algorithm selects action with highest value:

$$a_t^* = argmax_a \widehat{Q_t}(a) \tag{2.53}$$

Without any kind of exploration, we can lock onto a suboptimal action forever. In fact, if $argmax_a \widehat{Q_t}(a) \neq argmax_a Q(a)$ we will end up to repeat the same error again, again and again. This implies that greedy has a linear total regret.

ϵ-greedy algorithm

The ϵ-greedy algorithm acts greedily with probability $1 - \epsilon$ and randomly with probability ϵ. Even if ϵ is small, this method continues to explore forever, gaining a minimum regret at each step. As we have already seen before, if an algorithm continues to explore, it will end up with a linear total regret.

Decaying ϵ_t-greedy algorithm

The last naive idea is to use an ϵ-greedy policy, as described above, and decay the value of ϵ episode after episode. This very basic approach has a logarithmic asymptotic total regret. Unfortunately, achieving this result requires advance knowledge of the best actions to follow. Hence, even if very naive, this algorithm, theoretically, can reach the sublinear result that we were looking for.

2.5.3 Optimistic Initialisation

The idea behind this method is simple and very straightforward. We initialise $Q(a) = r_{max}$ then we let the system acts greedily.

$$a_t = argmax_a Q_t(a) \tag{2.54}$$

At the same time, we encourage the exploration of unknown values. The problem is that if we are unlucky, we risk locking onto suboptimal actions. Hence, this optimistic greedy idea still has a linear total regret.

2.5.4 Optimism in the face of uncertainty

Suppose we have three different probability distribution (Figure 2.10), one for each action (a_1, a_2 and a_3), to have a certain expected reward $Q(a_i)$. Let's assume that we have a deep knowledge of action a_3 a good knowledge of action a_2, but we do not know the $p(Q)$ of action one. This fact implies that action three and two will have longer tails than action one. At this point, we select the action that potentially can have a higher action-value. In the case of Figure 2.10, action 1 is the one with the lowest mean (between 1.2 and 1.5) but is also the one that potentially can the higher action-value (greater than 4). Hence, we select action 1 and based on the result we update $p(Q_1)$. We iterate this process until we have enough information to understand which action is the one with the highest $Q(a)$. Moreover, using Hoeffding's inequality, we can show that this algorithm achieves logarithmic asymptotic total regret without knowing the best actions to select.

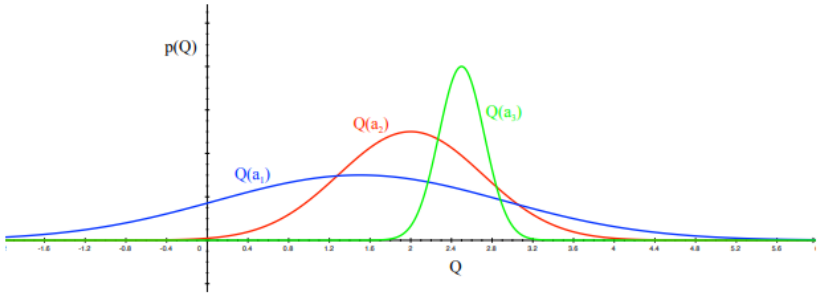


Figure 2.10: Example with three different distributions [8].

2.5.5 Probability Matching

Until now, we start with no assumptions about the reward distribution. Consider a distribution $p[Q|w]$ over action value-function with parameter w. Then, we compute the posterior distribution over w:

$$p[w|R_1, ..., R_t] \tag{2.55}$$

Finally, use posterior knowledge to guide exploration. Higher we are confident about the prior better will be the final result. In particular, probability matching selects action a according to the probability that a is the optimal action.

$$\pi(a|h_t) = P[Q(a) > Q(a'), \forall a' \neq a|h_t] \tag{2.56}$$

2.6 Deep Q-Networks

2.6.1 Introduction

All the solutions that we have seen so far imply storing a value for each state or state-action pair. These methods can be applied to small applications like the "Small Grid-World", where the number of states is relatively scanty. The difficulties arise when we try to apply reinforcement learning to large scale problems in which the state space is gigantic or even continuous. In these cases, it is impossible to use look-up tables, and we have to find alternative solutions. A possible answer to this problem could be to estimate the value function with a function approximator. Hence, if we denote with w the set of weights used by the approximator, we can rewrite the value function and the action-value function as:

$$\hat{v}(s, w) \approx v_\pi(s) \tag{2.57}$$

$$\hat{q}(s, a, w) \approx q_\pi(s, a) \tag{2.58}$$

Hence, given a certain state s, we can use a function approximator that returns $\widehat{v}(s, w)$. In the same way, we can obtain $\widehat{q}(s, a, w)$ starting from a couple (s, a). What often plays out is that we use the function approximator to predict not only an action-value but, given a state s, all the possible action-value function $\widehat{q}(s, a_m, w)$. Then, applying a softmax function, we can select greedily the next action to pick. To obtain these results, we can consider differentiable function approximators like Linear combinations of features or Neural networks. The problems with both methods are that they need iid and non-stationary data. In an RL setting, obtain independent samples is quite irrealistic since where the agent will be in the next step is strongly dependent on where it is now. Moreover, due to the randomness of the world is very complicated to obtain stationary data. Having said that, we can mitigate these problems, but we will see further down the line.

2.6.2 Approximation

Gradient Descent

Let $j(w)$ be a differentiable function of parameter vector w. To find a local minimum of $j(w)$, we can use minus the direction of the gradient of w.

$$\nabla w = -\frac{1}{2}\alpha \nabla_w j(w) \tag{2.59}$$

where alpha works like a learning rate parameter. At this point, we assume that we a sort of oracle that told us the value function $v_\pi(s)$. The goal is to find a parameter vector w that minimise the error between the approximate value function $\widehat{v}(s, w)$ and $v_\pi(s)$. Now if we define j as the expected mean-square error between v and $\widehat{v}$, we obtain:

$$j(w) = E_\pi[(v_\pi(s) - \widehat{v}(s, w))^2] \tag{2.60}$$

If we try to apply gradient descent to minimise this error we find:

$$\Delta w = \alpha E_\pi[(v_\pi(s) - \widehat{v}(s, w))\nabla_w \widehat{v}(s, w)] \tag{2.61}$$

since all the values are already expectations and not random variable anymore, we can rewrite:

$$\Delta w = \alpha(v_\pi(s) - \widehat{v}(s, w))\nabla_w \widehat{v}(s, w) \tag{2.62}$$

Where α represents the step size, $v_{pi} - \widehat{v}$ is the prediction error and $\nabla_w \widehat{v}$ indicates the direction to follow to reduce the error.

2.6.3 Linear value function

A possible way to represent the state of the agent could be using a feature vector:

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix} \tag{2.63}$$

Each feature describes the environment in which the agent is actually playing. For instance, each predictor can represent a piece and pawn configurations in chess or go game. At this point, we can rewrite our approximated value function in term of the features and weight vector.

$$\widehat{v}(s, w) = x(s)^T w \tag{2.64}$$

Replacing this term in the $j(w)$ equation what we obtain is:

$$j(w) = E_\pi[(v_\pi(s) - x(s)^T w)^2] \tag{2.65}$$

Since this function is quadratic with respect to w, if we apply gradient descent, we are sure to end up in a global minimum. In fact, in this case, there is no chance to end up at a local minimum. Moreover, the update rule is especially easy since:

$$\nabla_w \widehat{v}(s, w) = x(s) \tag{2.66}$$

$$\Delta w = \alpha(v_\pi(s) - \widehat{v}(s, w))x(s) \tag{2.67}$$

Looking at this new update rule it's clear that we update only the weights of the features that are different from zero.

2.6.4 Prediction

Until now, we have supposed to have an oracle that gives us the true value of $v_\pi(s)$. In real applications, the environment does not provide that kind of information but only rewards. Hence, as already seen previously, we replace $v_\pi(s)$ with a target value. For instance, in Monte-Carlo learning, we can use the expected return G_t:

$$\Delta w = \alpha(G_t - \widehat{v}(s, w))\nabla_w \widehat{v}(s, w) \tag{2.68}$$

The convergence properties of this algorithm are remarkable both in linear and in non-linear settings. In fact, MC evaluation always converges to a local optimum. Another possibility is to use $TD(\lambda)$:

$$\Delta w = \alpha(G_t^\lambda - \widehat{v}(s, w))\nabla_w \widehat{v}(s, w) \tag{2.69}$$

In a linear setting, using an On-policy algorithm, $TD(\lambda)$ can converge very close to a global optimum. How far from this result depends on the hyperparameters choice, like the discount factor and the step-size α. While $TD(\lambda)$ applied to Off-policy methods not converge neither in a Linear nor in a Non-linear setup.

2.6.5 Control

Control works like prediction but, we replace the value function $\widehat{v}(s, w)$ with the action-value function $\widehat{q}(s, a, w)$. In this way, we can continue to deal with model-free algorithms. The updated formulas are really similar to the above one.

- For MC:

$$\Delta w = \alpha(G_t - \widehat{q}(s, a, w))\nabla_w \widehat{q}(s, a, w) \tag{2.70}$$

- For $TD(\lambda)$:

$$\Delta w = \alpha(q_t^\lambda - \widehat{q}(s, a, w))\nabla_w \widehat{q}(s, a, w) \tag{2.71}$$

The convergence results are summed up in the following table:

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗

Figure 2.11: Linear MC and Sarsa chatters around near-optimal value function [8].

2.6.6 Batch reinforcement learning

What we have done until now is to use a new sample to update our value or action-value function and then throw it away. This means that we only look forward without caring too much about past experience. This not so efficient, especially if we try to apply a gradient descent method. In particular, GD works well but is not sample efficient. Hence, the idea is to use all the agent's experience to fit the value function. A possible solution is to use the least-squares method. In this case, we store all the state-value pairs in a vector D:

$$D = < s_1, v_1^\pi >, ..., < s_n, v_n^\pi > \tag{2.72}$$

Then we look for the vector w that minimise the sum-squared error between $\widehat{v}(s_t, w)$ and v_t^π:

$$LS(w) = \sum_t (v_t^\pi(s_t) - \widehat{v}(s_t, w)) \tag{2.73}$$

At this point, to solve the problem of iid data, we can introduce the experience replay. The main idea behind this method is to sample data from experience. In

this way, data can come from different points of the episode or even be part of different episodes. Finally, we can apply the stochastic gradient descent update:

$$\Delta w = \alpha(v_\pi(s) - \widehat{v}(s,w))\nabla_w \widehat{v}(s,w) \tag{2.74}$$

This solution converges to:

$$w^\pi = argmin_w LS(w) \tag{2.75}$$

2.6.7 DQN

Deep Q-Networks (DQN) uses deep neural networks, experience replay and fixed Q-targets to obtain the approximated action-value functions $Q(s,a;w_i)$. This method, thanks to these ingredients mixed all together, is very stable and doesn't blow up as TD and MC learning with non-linear functions. DQN is a variation of the Q-learning algorithm, as we can see from the error function:

$$L_i(w_i) = E_{s,a,r,s'}[(r + \gamma\, max_{a'} Q(s',a';w_i^-) - Q(s,a;w_i))^2] \tag{2.76}$$

We have already seen experience replay that samples random mini-batches of transitions (s, a, r, s') from the replay memory, but what is really new in this formula is the $Q(s',a';w_i^-)$ term. In this case, the Q-learning target is not retrieved using the same weights w but using some old weights. For instance, if w are weights obtained after 1000 epochs, the w^- can be the weights obtained at the 900^{th} epoch. In this way, we decorrelate the values predicted, increasing the performance of the method. Experience replay and fixed Q-targets are two game-changers that can dramatically boost the performance of DQN. In Figure 2.12, you can see the impact of these two methods applied to some Atari games.

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Figure 2.12: Performances on Atari games [8].

2.7 Policy gradient methods

2.7.1 REINFORCE algorithm

Let's consider a neural network (NN), with a set of weights θ, that produces as output our stochastic policy $\pi_\theta(a|s)$. Then from this probability distribution, we sample an action a and, it will become the input for the environment. We repeat this procedure T times to end up with a trajectory:

$$\tau = (s_1, a_1, ..., s_T, a_T) \tag{2.77}$$

Hence, the probability to obtain exactly the trajectory τ can be computed as:

$$p_\theta(\tau) = p(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_t^T \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.78}$$

At this point, we aim to train the set of weights until we obtain the values θ^* that maximise the expected return over the entire trajectory.

$$\theta^* = argmax_\theta E_{p_\theta(\tau)}[\sum_t r(s_t|a_t)] \tag{2.79}$$

If we indicate with $J(\theta)$ the expected return over the trajectory τ, we can make explicit the expected value:

$$J(\theta) = E_{p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.80}$$

Finally, to estimate θ^*, we can compute the gradient with respect to θ of this new function $J(\theta)$ and perform gradient ascent.

$$\nabla_\theta J(\theta) = \nabla_\theta \int p_\theta(\tau)\, r(\tau)\, d\tau = \int \nabla_\theta p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.81}$$

As the integral is a linear operator, we can bring the gradient inside the integral. At this point, we should differentiate the transition probability matrix (TPM), but that we do not know it since we do not have information about the environment. To overcome this problem, we can try to exploit the following convenient identity:

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = p_\theta(\tau) \nabla_\theta log\, p_\theta(\tau) \tag{2.82}$$

replacing this term in the previous equation we obtain:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau)\, r(\tau)\, d\tau = \int p_\theta(\tau) \nabla_\theta log\, p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.83}$$

Based on the previous definition of $p_\theta(\tau)$, we can apply the log function; in this way the productoria becomes a sommatoria.

$$log\, p_\theta(\tau) = log\, p(s_1) \sum_t^T log\, \pi_\theta(a_t|s_t) + log\, p(s_{t+1}|s_t, a_t) \tag{2.84}$$

Thanks to this result, we can finally differentiate this quantity because the first and the last term does not depend on θ anymore. Only $\sum_t^T log\, \pi_\theta(a_t|s_t)$ "survives" and this term does not include the TPM. So, after some mathematical tricks, we obtain:

$$\nabla_\theta J(\theta) = E_{p_\theta(\tau)}[(\sum_t^T \nabla_\theta\, log\, \pi_\theta(a_t|s_t))(\sum_t^T r(s_t, a_t))] \tag{2.85}$$

Computing this quantity is relatively easy since the rewards come from the environment, while $\pi_\theta(a, s)$ is the output of our NN.

We have to highlight that this quantity is similar to the maximum likelihood concept but with the reward modification. In fact, in the policy gradient case we:

- increase the probability to see a certain trajectory if the total reward is positive
- decrease the probability to see a certain trajectory if the total reward is negative.

In other words, this is the formalization of the "trial and error" approach since good actions are made more likely while bad actions are made less likely. We can finally write the update rule of the REINFORCE algorithm:

1. sample τ_i from the policy $\pi_\theta(a_t|s_t)$
2. compute $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i^N (\sum_t^T \nabla_\theta\, log\, \pi_\theta(a_t^i|s_t^i))(\sum_t^T r(s_t^i, a_t^i))$
3. update $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
4. Repeat from step 1.

The first problem with this algorithm starts from $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i^N \nabla_\theta\, log\, \pi_\theta\, r(\tau)$. In fact, depending on the samples obtained, this method can end up with very different distributions. Moreover if the $r(\tau) = 0$, the trajectory does not affect at all the final result. This fact implies that the REINFORCE algorithm is affected by a high variance problem.

Reducing Variance

To solve this variance problem, we can try to exploit two different concepts:

- Causality
- Baselines

The causality principle implies that the policy at time t' can not affect reward at time t when $t < t'$. This sentence looks quite obvious; the action chosen in the present can not alter the reward obtained in the past. So, instead of using the Monte-Carlo approach seen until now, we can use an approximate version of the action-value function. In fact, we consider only the rewards from time t' to the end of the episode.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \, log\, \pi_\theta(a_t^i|s_t^i)(\sum_{t=t'}^{T} r(s_t^i, a_t^i)) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \, log\, \pi_\theta(a_t^i|s_t^i)\widehat{Q}_{i,t} \tag{2.86}$$

Where $\widehat{Q}_{i,t}$ is also called "reward to go". This method is still unbiased but can lower the variance since the final result depends on a lower number of actions, transactions and rewards. This fact is already seen in the differences between the MC and the TD algorithm.

Another possibility is to use Baselines. In general, it's not true that if we find a negative return for a trajectory, we have to make that path less likely. In fact, in some cases, we have only negative rewards so, all the trajectories can only have a negative return. So, the main idea behind baselines is to centre the returns. In this way, we increase the probability of the paths that have a return higher than the average and vice-versa. To do that we subtract to the reward a constant $b = \frac{1}{N}\sum_i^N r(\tau)$. With this solution, policy gradient can work even in the case where all the rewards have the same sign. At this point, it's important to highlight that we are allowed to that because it will not change the mean in expectation.

$$E[log\, \pi_\theta(\tau)b] = \int p_\theta(\tau)\nabla_\theta \, log\, p_\theta(\tau)b\, d\tau = \int \nabla_\theta \, p_\theta(\tau)b\, d\tau \tag{2.87}$$

We can still rewrite this equation taking the gradient and the b outside the integral. This is possible because b does not depend on τ and the integral is a linear operator.

$$\int \nabla_\theta \, p_\theta(\tau)b\, d\tau = b\nabla_\theta \int p_\theta(\tau)d\tau = b\nabla_\theta 1 = 0 \tag{2.88}$$

The integral of a probability density is always equal to one and the gradient of a constant is zero by definition. So, subtracting a baseline is unbiased in expectation and can help to reduce the variance. Average reward b is not the best baseline but in practice, it works really well.

2.8 SAC vs DDPG

2.8.1 Introduction

In this section, we will compare two different reinforcement learing approach: "Soft Actor Critic" (SAC) by Haarnoja et al. and "Deep Deterministic Policy Gradient" (DDPG) by Silver et al.. Before talking about their differences, let's talk about what they have in common.

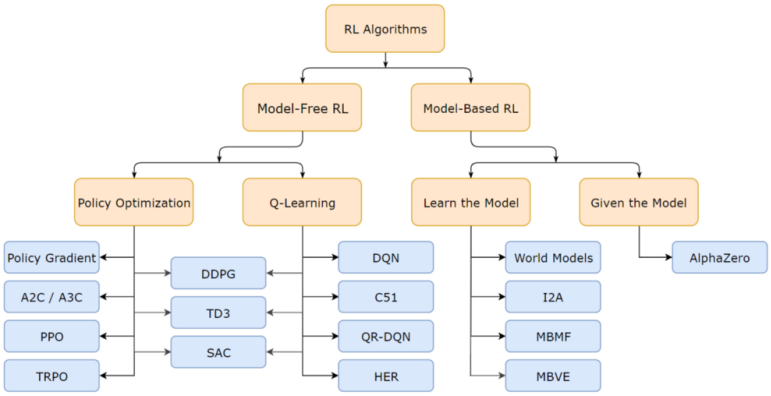


Figure 2.13: RL algorithms panoramic [12].

As shown in Figure 2.13, they are both model-free reinforcement learning algorithm; this means that we do not know the full dynamic and the reward function of that Markov decision process (MDP). Hence, Model-free RL ignores the model and care less about the inner working. It uses sampling and simulation to understand how to maximize the cumulative reward. This operation can be computational and time expensive, especially if we have to simulate it in a real-world environment. Both in simple and in complex settings, Model-free RL requires million of steps of data collections to perform accurately. At this point, we can make another distinction between the on-policy and off-policy learning algorithm

- In the former case, to compute a new policy step, we have to collect novel samples from the freshest policy. This behaviour is sample inefficient since all the past experience is thrown away. To mitigate this problem, we often run more than one agent in parallel.
- In the letter case, we reuse the past experience to train the model. The main idea under this method is that an agent can learn how to behave by looking at other agents. Even if the policy used by others is worst than ours, we can learn from their mistakes. Unfortunately, the combination of off-policy

learning and high-dimensional, non-linear function approximation with neural networks presents a major challenge for stability and convergence.

Moreover, Model-free RL is highly sensitive to the hyperparameter choice and, the number of parameters to be tuned can not be neglected. Coming back to SAC and DDPG, they use both policy gradient and q-learning to obtain the final action prediction. The main idea under policy gradient is very naive so, let's consider a neural network (NN), with a set of weights θ, that produces as output our stochastic policy $\pi_\theta(a|s)$. Then from this probability distribution, we sample an action a and, it will become the input for the environment. We repeat this procedure T times to end up with a trajectory:

$$\tau = (s_1, a_1, ..., s_T, a_T) \tag{2.89}$$

Hence, the probability to obtain exactly the trajectory τ can be computed as:

$$p_\theta(\tau) = p(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_t^T \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.90}$$

At this point, we aim to train the set of weights until we obtain the values θ^* that maximise the expected return over the entire trajectory.

$$\theta^* = argmax_\theta E_{p_\theta(\tau)}[\sum_t r(s_t|a_t)] \tag{2.91}$$

If we indicate with $J(\theta)$ the expected return over the trajectory τ, we can make explicit the expected value:

$$J(\theta) = E_{p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.92}$$

Finally, to estimate θ^*, we can compute the gradient with respect to θ of this new function $J(\theta)$ and perform gradient ascent.

$$\nabla_\theta J(\theta) = \nabla_\theta \int p_\theta(\tau)\, r(\tau)\, d\tau = \int \nabla_\theta p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.93}$$

As the integral is a linear operator, we can bring the gradient inside the integral. At this point, we should differentiate the transition probability matrix (TPM), but that we do not know it since we do not have information about the environment. To overcome this problem, we can try to exploit the following convenient identity:

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = p_\theta(\tau) \nabla_\theta log\, p_\theta(\tau) \tag{2.94}$$

replacing this term in the previous equation we obtain:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau)\, r(\tau)\, d\tau = \int p_\theta(\tau) \nabla_\theta log\, p_\theta(\tau)\, r(\tau)\, d\tau \tag{2.95}$$

Based on the previous definition of $p_\theta(\tau)$, we can apply the log function; in this way the productoria becomes a sommatoria.

$$log\, p_\theta(\tau) = log\, p(s_1) \sum_t^T log\, \pi_\theta(a_t|s_t) + log\, p(s_{t+1}|s_t, a_t) \tag{2.96}$$

Thanks to this result, we can finally differentiate this quantity because the first and the last term does not depend on θ anymore. Only $\sum_t^T log\, \pi_\theta(a_t|s_t)$ "survives" and this term does not include the TPM. So, after some mathematical tricks, we obtain:

$$\nabla_\theta J(\theta) = E_{p_\theta(\tau)}[(\sum_t^T \nabla_\theta\, log\, \pi_\theta(a_t|s_t))(\sum_t^T r(s_t, a_t))] \tag{2.97}$$

Computing this quantity is relatively easy since the rewards come from the environment, while $\pi_\theta(a, s)$ is the output of our NN.

We have to highlight that this quantity is similar to the maximum likelihood concept but with the reward modification. In fact, in the policy gradient case we:

- increase the probability to see a certain trajectory if the total reward is positive
- decrease the probability to see a certain trajectory if the total reward is negative.

In other words, this is the formalization of the "trial and error" approach since good actions are made more likely while bad actions are made less likely. This general idea is a good starting point but can be still improved. Computing the reward in this way introduce a high variance that adversely affects the performance of the algorithm. So, instead of using the exact reward computed by the experience, we can fit a value function $\widehat{V}_\theta^\pi$. Hence, given a determined state s, we can approximate the reward sum as $\widehat{V}_\theta^\pi(s)$. This improvement dramatically increases the converging time but, it's not enough and this becomes immediately clear when we look at the expression:

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta\, log\, \pi_\theta(a_t|s_t)(r(s_t, a_t) + \widehat{V}_\theta^\pi(s')) \tag{2.98}$$

This general idea is a good starting point but can be still improved. Computing the reward in this way introduce a high variance that adversely affects the performance of the algorithm. So, instead of using the exact reward computed by

the experience, we can fit a value function $\widehat{v}_\theta^\pi$. Hence, given a determined state s, we can approximate the reward sum as $\widehat{v}_\theta^\pi(s)$. This improvement dramatically increases the converging time but, it's not enough and this becomes immediately clear when we look at the expression:

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \, log \, \pi_\theta(a_t|s_t)(r(s_t, a_t) + \widehat{V}_\theta^\pi(s')) \tag{2.99}$$

What we notice is that when the second term $r(s_t, a_t) + \widehat{V}_\theta^\pi(s')$ is positive, the actions a_t are made more probable as already said before. This behaviour, in some cases, can not be the optimal one. Let's suppose that we start from a state s where $\widehat{V}_\theta^\pi(s)$ is a high positive value. Then we choose an action a_t and we end up in state s' where $\widehat{V}_\theta^\pi(s')$ is a quite small non-negative value as $r(s_t, a_t)$. If this happens, this means that action a_t even if it gets a positive reward it's not a good action and, we should lower the probability of seeing that action again. This because there are for sure other action a_t' from state s that will lead to better states s'' or rewards $r(s_t, a_t')$. For this reason, we can rewrite the aforementioned equation:

$$\nabla_\theta J(\theta) \approx \sum_t \nabla_\theta \, log \, \pi_\theta(a_t|s_t)(r(s_t, a_t) + \widehat{V}_\theta^\pi(s') - \widehat{V}_\theta^\pi(s)) \tag{2.100}$$

In this case the term $r(s_t, a_t) + \widehat{V}_\theta^\pi(s') - \widehat{V}_\theta^\pi(s)$ tells us how much better is take action a_t with respect to the average. This formulation is also known as the Actor-Critic method. As an agent takes actions and moves through an environment, it learns to map the observed state of the environment to two possible outputs:

- the best action to take, given by our policy approximator.
- the expected reward has taken that particular action.

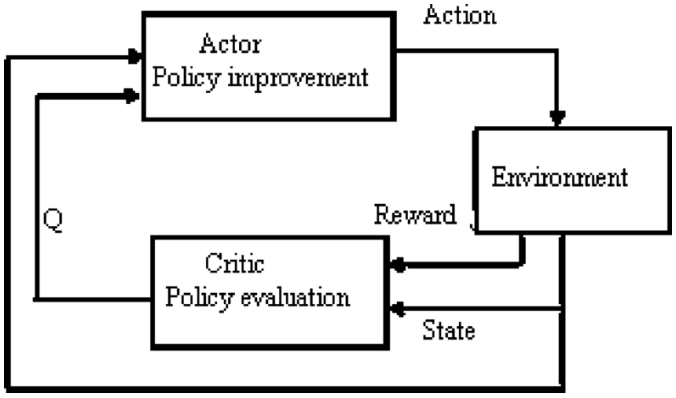


Figure 2.14: Actor-Critic mechanism [13].

The other algorithm that is crucial in both SAC and DDPG is Q-learning. This method is an off-policy learning method that exploits the Bellman optimality equation to update the action-value function $Q(S, A)$. It's based on two different policy: a behaviour (μ) and a target (π) one. Starting in state S, the new action A is chosen ϵ-greedy following the behaviour policy. Taking action A, we end up in a new state S'. At this moment, the target policy comes into action. This means that the new action A' is chosen greedily with respect to Q(S, A):

$$\pi(S') = argmax_a Q(S', a) \tag{2.101}$$

This implies that the new Q-learning target becomes:

$$R_t + \gamma \, max_a(Q(S', a)) \tag{2.102}$$

while the Q-learning error can be rewritten as:

$$R_t + \gamma \, max_a(Q(S', a)) - Q(S, A) \tag{2.103}$$

Having said that the final Q-learning update rule:

$$Q(S, A) = Q(S, A) + \alpha(R_t + \gamma \, max_a(Q(S', a)) - Q(S, A)) \tag{2.104}$$

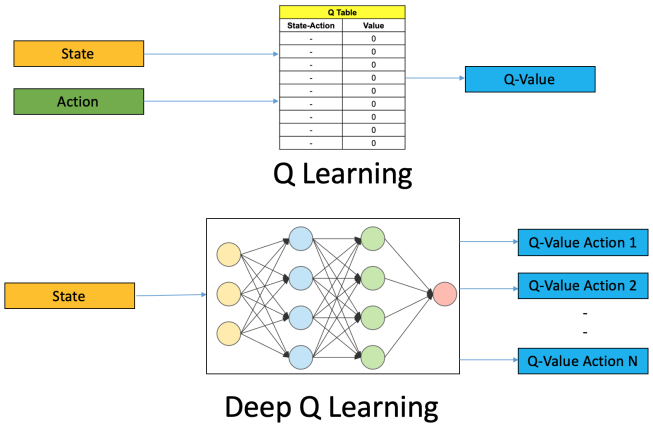


Figure 2.15: Q-learning mechanism [14]

Before looking into the details of both the algorithms, we have to highlight that both the methods are developed for solving, in an efficient way, continuous space and action RL environments. In particular, SAC aims to overcome some limitations of the DDPG solution.

2.8.2 SAC

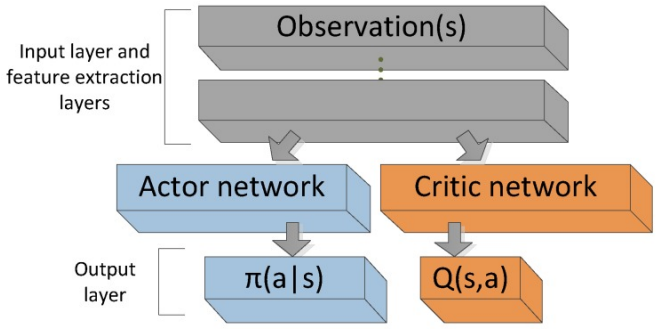


Figure 2.16: Sac structure [15]

The key features of the SAC algorithms are the following:

1. it optimizes a stochastic policy in an off-policy way
2. it uses entropy regularization to find the right trade-off between exploration and exploitation
3. it uses the clipped double-Q trick to mitigate the positive bias problem

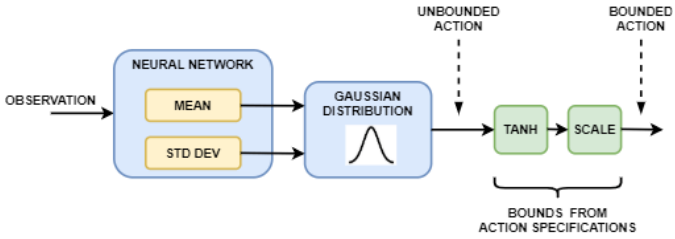


Figure 2.17: Compute the optimal action SAC [16].

Let's start to analyse all these three features one by one. Starting from the first point, how the next action is chosen is shown in Figure 2.17. As already said before, this algorithm uses the past experience collected in order to compute the next gradient step. What the policy network produce in output is the mean and the standard deviation of a normal distribution. From this distribution, we sample the following action to execute. This solution is explicitly designed for a continuous actions space environment but can be modified to work in a discrete one. This stochasticity does not consider only the most possible action so, it can help to explore all the possible moves. But to provides a substantial improvement

in exploration and robustness, SAC introduces a new term called entropy. Entropy is a scientific concept that is most commonly associated with a state of disorder, randomness, or uncertainty. In Probability, this concept is referred to how sure we are about a certain event. Let's consider an unfair coin where the probability of obtaining head is 0.9 and tail 0.1. In this case, the entropy is low since we know that most of the time, we will obtain head. Whilst, in the case of a fair coin, the uncertainty of the final result translates into a high entropy. So, let x be a random variable with probability mass or density function P. The entropy is computed from its distribution according to:

$$H(P) = -E_{x\sim P}[log(P(x))] \tag{2.105}$$

Maximizing this quantity means having an action distribution, for each state, similar to a uniform one. This solution ensures that the algorithm doesn't stick in a global minimum but explores all the possible actions and select the best ones. Hence, the policy is trained to maximize the trade-off between expected return and entropy. This changes the RL problem to:

$$\theta^* = argmax_\theta E_{p_\theta(\tau)}[\sum_t r(s_t|a_t) + \alpha H(\pi(\cdot|s_t))] \tag{2.106}$$

Where α is also called the temperature parameter. This hyperparameter can be fixed or can be learnt using an automatic gradient-based tuning method that adjusts the expected entropy over the visited states to match a target value. Having a high α means encourage exploration while having a low α value means exploitation. Usually, we start with a relatively high α to let the algorithm explore and then we slowly decrease it. This parameter for some aspect is similar to ϵ of the ϵ-greedy policy. SAC can suffer from brittleness to the temperature so, the gradient-based solution, can mitigate this problem. Another very common problem in q-learning literature is the overestimation of the action-value function. A usual solution to this problem is using double q-learning that involves using two separate Q-value estimators (ϕ and ϕ_{tar}). Using these independent estimators, we can obtain unbiased Q-value estimates. To further improve this result, we can double the number of networks ending up with ϕ_1, $\phi_{tar,1}$ and ϕ_2, $\phi_{tar,2}$. In this way, we can reduce the overestimation of $Q(s,a)$ taking the lowest value between $Q_{\phi_{tar,1}}$ and $Q_{\phi_{tar,2}}$. Putting it all together, the loss functions for the Q-networks in SAC are:

$$L(\phi_i) = E[(Q_{\phi_i}(s,a) - (min_j\, Q_{\phi_{tar,j}}(s', \widetilde{a_\theta}) + H(\widetilde{a_\theta})))^2] \quad \forall i \in [1,2] \tag{2.107}$$

where $\widetilde{a_\theta} \sim \pi_\theta(\cdot|s')$. Then, the target networks are updated once per main networks update by polyak averaging:

$$\phi_{tar_i} = (1-\tau)\phi_{tar_i} + \tau\phi_i \quad \forall i \in [1,2] \tag{2.108}$$

Finally, The policy is thus optimized according to:

$$max_\theta E_{p_\theta(\tau)}[min_j\, Q_{\phi_j}(s', \widetilde{a_\theta}) + \alpha H(\widetilde{a_\theta})] \tag{2.109}$$

PseudoCode

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta \left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 2.18: SAC pseudo code [17].

2.8.3 DDPG

DDPG can be seen as the extension of DQN to continuous action spaces. DDPG exploits off-policy data and the Bellman equation to learn a q-function $Q(s, a)$ and uses this function to learn the policy. For this reason (as shown in Figure 2.19), this is an Actor-Critic algorithm. This approach, even if connected to the DQN, aims to overcome the discrete action space problem. In DQN, to decide the new action to follow we compute $a^* = argmax_a(Q(s', a))$. But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. A naive solution is to discretize the action space but, this will raise two different problems: Discover a smart way to divide the continuous space into bins that effectively describe the environment The number of bins becomes another hyperparameter to tune. Another possibility is to exploit the fact that the action-value function $Q(s, a)$ is now continuous and differentiable with respect to the action a. This solution allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$. Finally instead of computing $max_a Q(s, a)$ we can approximate this quantity with $Q'(s, \mu(s))$. To update the critic network, we aim

to minimise the MSE loss:

$$L(\phi_i) = E[(Q_{\phi_i}(s,a) - (r(s,a) + Q_{\phi_{tar}}(s', \mu_{\theta_{tar}}(s')))^2] \tag{2.110}$$

As DQN, DDPG uses the target networks ϕ_{tar} and θ_{tar}. These networks are introduced to avoid a moving target. Having a moving target in a supervised learning task can adversely affect the performance of the model. For this reason, the target networks are not updated for epochs in order to output the same result for the same input. In the DDPG implementation, the target networks are instead slightly updated every episode using the Polyak rule (the same already seen in SAC). Finally, we want to learn a deterministic policy $\mu(s)$ that maximise $Q(s,a)$. To that, we can apply a gradient ascent strategy with respect to the parameter θ:

$$max_\theta E[Q_\phi(s, \mu_\theta(s))] \tag{2.111}$$

To make the algorithm more robust and allow the right trade-off between exploration and exploitation, we introduce noise into the input space. Different solutions can be used, from a simple Gaussian noise to the time-correlated OU noise. As in SAC, in the first stages of the algorithm, we prefer to try to explore the action space performing new solutions. Then, in order to improve the final result, we can decrease this noise episode per episode, preferring exploitation over exploitation.

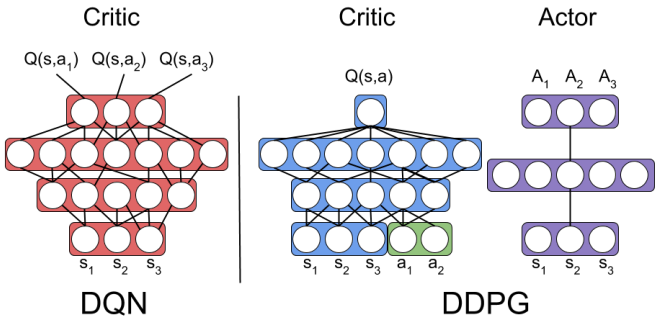


Figure 2.19: DQN vs DDPG

PseudoCode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

Figure 2.20: DDPG pseudo code [18].

2.8.4 Comparison in different environments

Walker2d-v2

The Walker2d environment consists of a humanoid walker trying to move forward. Since it is a 2D environment, the walker can only fall forwards and backwards. When it happens, the episode ends. Anyway, to avoid too long episodes the maximum length is set to 1000 steps. In this environment, there are 17 observations and 6 actions for the walker: the 17 observations include velocities of different parts of the body and joint angles, while the 6 actions represent signals on how to move the joints of the torso and the legs. To evaluate the walker's performance, the reward function is the current velocity plus a constant bonus for being alive minus a shaping term. This last term is the squared sum of all the actions, and it is introduced to smooth the reward gradient.

Results of Sac and DDPG on Walker2d-v2

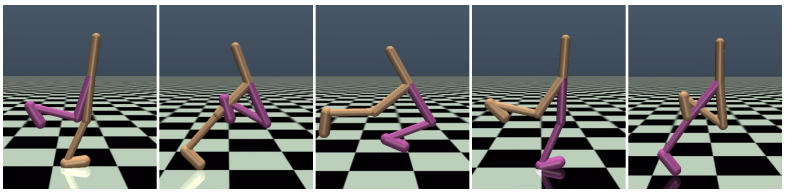


Figure 2.21: Walker environment

Table 2.1: Result of DDPG and SAC on Walker2d[19].

Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	2292.266	13.97	1M	150
DDPG	1387.591	736.955	1M	208

Table 2.2: Result of DDPG and SAC on Walker2d[20].

Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	2052.646	13.631	150000	150
DDPG	1954.753	368.613	149152	155

LunarLander-v2

The landing pad starts always at coordinates (0,0). Coordinates are the first two numbers in the state vector. The reward for moving from the top of the screen to the landing pad and zero speed is about 100 $\sim$ 140 points. If the lander moves away from the landing pad it loses reward. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Landing outside the landing pad is possible. Firing the main engine is -0.3 points for each frame. The environment is solved if we achieved 200 points in the last 100 episodes. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real values vector from -1 to +1. First controls main engine, -1..0 off, 0..+1 throttle from 0.5 to 1.0 power. The engine can't work with less than 0.5 power. Second value -1.0..-0.5 fire left engine, +0.5..+1.0 fire right engine, -0.5..0.5 off [21].

Results of Sac and DDPG on LunarLander-v2

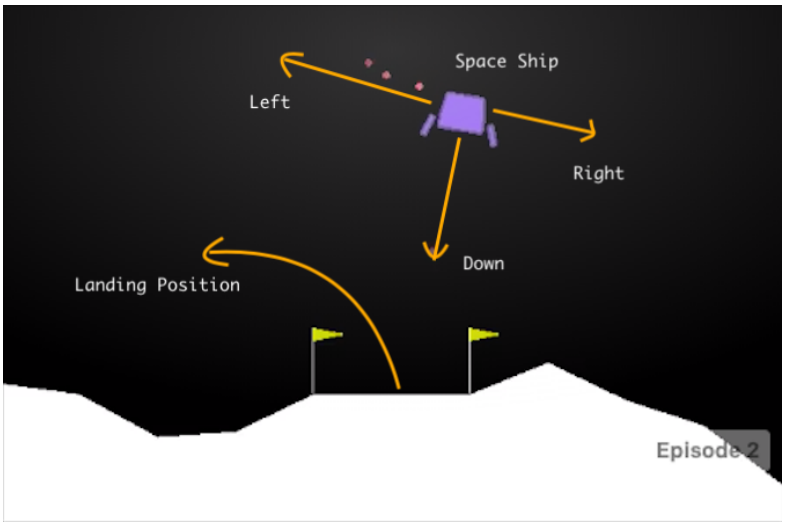


Figure 2.22: LunarLander environment [21].

Table 2.3: Result of DDPG and SAC on LunarLander[19].

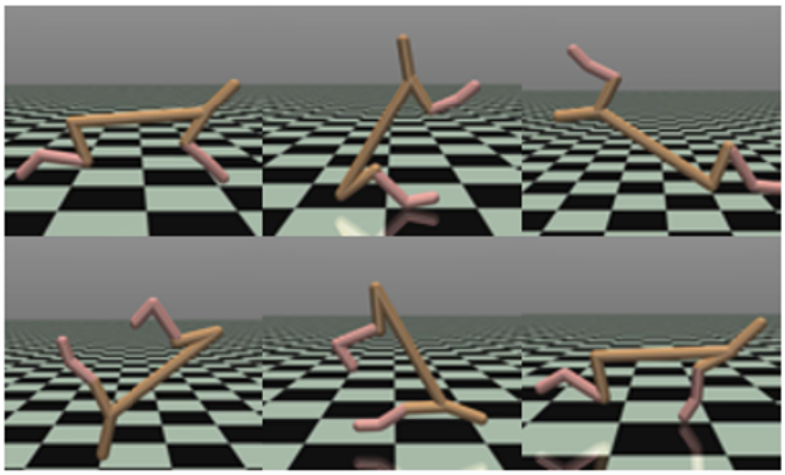
Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	260.390	65.467	500k	672
DDPG	230.217	92.372	300k	556

Table 2.4: Result of DDPG and SAC on LunarLander[20].

Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	269.783	57.077	149852	709
DDPG	244.566	75.617	149531	660

HalfCheetah-v2

The half-cheetah task has a 17-dimensional state space consisting of joint positions, joint velocities, horizontal speed, angular velocity, vertical speed and relative height. Actions have six dimensions vector from -1 to +1 and are accelerations of joints. The reward function r(s) = max($\frac{\Delta x}{10}$,0) where Δx is the horizontal speed to encourage forward motion.



Results of Sac and DDPG on HalfCheetah-v2

Table 2.5: Result of DDPG and SAC on HalfCheetah[19].

Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	2792.170	12.088 1M	500k	150
DDPG	2078.325	208.379	1M	150

Table 2.6: Result of DDPG and SAC on HalfCheetah[20].

Algorithm	*Mean reward*	*Std reward*	*Train steps*	*Test episodes*
SAC	3330.911	95.575	150000	150
DDPG	2549.452	37.652	150000	150

2.9 Conclusions

Until now, we have seen two different RL models that can be applied to our problem: model-based and model-free. The first approach can not be used mainly for two different reasons regarding the input data: The amount of input is incredibly low The input data type can not accurately describe the real world. It's impossible to model real-world behaviour using only asset prices. For this reason, model-free seems better to face this kind of problem. The low amount of data still be a problem but, we will try to exploit the inductive biases of the machine learning architectures used to mitigate this problem. We will look in detail at this problem in the following chapter 3.4. At this point, we have analysed two very competitive model-free RL algorithms: SAC and DDPG. We have tested our algorithm in

different environments to look at which is the most suitable for our case. We have preferred to report results from different sources to be sure to select the most powerful model. Based on [20] and [19], we can see that SAC outperforms DDPG in every environment. SAC seems to be even more stable with a standard deviation of the final reward significantly lower than the one obtained by DDPG. In the literature, there are papers that already use DDPG for this kind of task as we will see in chapter 4. For this reason, our experiments will mainly be based on this algorithm.

Chapter 3

Transformer

3.1 Introduction

This chapter wants to give a very brief introduction to what a neural network is and how it works. Then we will focus on the Transformer architecture, the model that we will use in our experiments. The aim of this chapter wants to be a detailed introduction to the tools used but not an exhaustive explanation of how these algorithms work. This chapter is mainly based on the book of Abhijit Gosavi [10] and Zhang [22] and the work of Ashish Vaswani [23], [24]. Finally, we will illustrate the reasons why Transformer was prefered over other ML architecture like CNN or LSTM. This work is done studying the equivariance and invariance properties of these tools [25], [26].

3.2 Neural networks introduction

There are several ways in which it is possible to implement a non-linear regressor or classificator, but a backpropagation method seems what it is needed to solve this problem. Backpropagation needs for every possible entry tuple an exit value in order to compute the loss function. It is then minimized with a gradient-method as it's shown in the least-squared approach. This is the basic idea behind neural network (NN). For the sake of simplicity and the objectives of this thesis, we will introduce NN in a regression setting. Very similar calculations can be computed for a classification task. These kind of networks are able to optimally recognize various types of patterns, and they are used in different fields such as regression. However, it is necessary to tune several hyperparameters to obtain an adequate result. NN are essentially based on nodes:

- Input nodes, which are in finite number, that usually equals the number of

inputs of the function. The set of all input nodes constitutes the so-called input layer.

- Hidden nodes make up that part of the NN that introduces a non-linearity into the model. They are organized in one or more layers. There are no fixed rules to select the correct number of hidden nodes or layers: only a "trial and error" approach can find an appropriate solution to problems (for our purpose, one hidden layer is enough but the problem of the number of nodes remains).
- Output node, which can be one or more depending on the type of task we are facing.

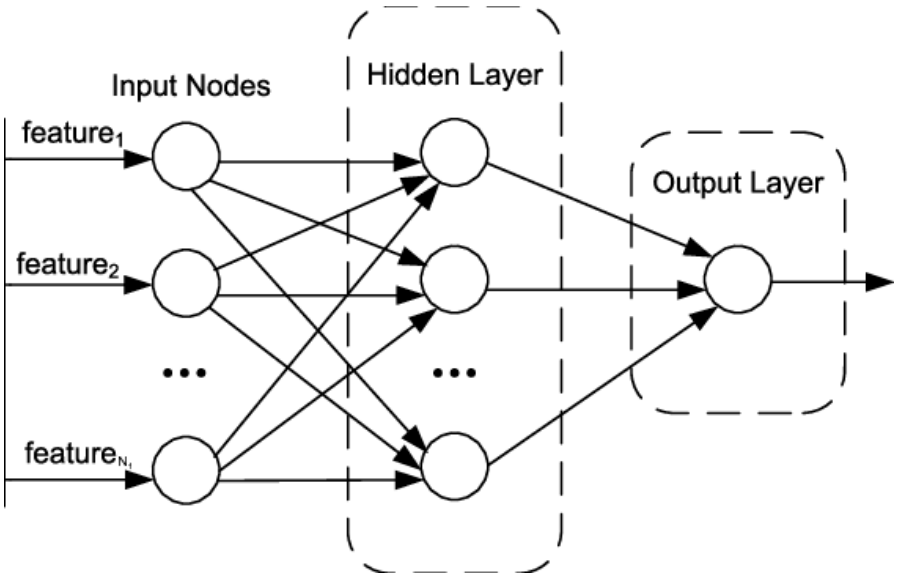
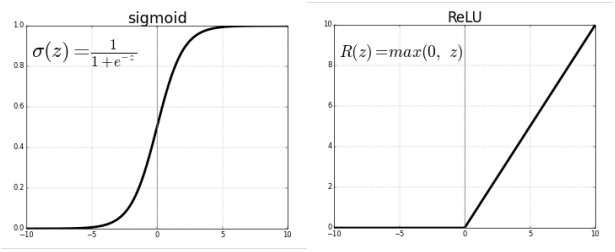


Figure 3.1: Standard visualization of feed forward linear network.

It's important to emphasize (as shown in the Figure 3.1) that every node of the input node is fully linked with the nodes of the hidden layer and, in turn, every node of hidden layer is connected with the output node. Each connection is only in one direction and it's associated with a scalar value called weight. From now on, we will denote by $w(i,h)$ the weight which joins the i_{th} input layer with the h_{th} hidden one. Indeed, we will denote by $x(h)$ the weight on the link from the h_{th} hidden node to the output node. So, every node has an internally stored value: for the input layer, we have the input values of dataset, while for the hidden and the output node we can compute this value using w and x. Indicating with $u(j)$ the value of the j_{th} input node, we can compute the vector $\hat{v}$ as:

$$\hat{v}(j) = \sum_{j=1}^{I} w(i,j)u(j) \tag{3.1}$$

This doesn't exactly match the value of hidden nodes, because, at this point, we can introduce the non-linearity in our model. Different non-linear activation functions can be used for this purpose, but the most popular ones are the sigmoid and the ReLU function.



Computed the value v of each hidden node, the output value of the NN is given by:

$$o = \sum_{h=1}^{H} x(h)v(h) \tag{3.2}$$

At this point, the backpropagation algorithm is used to minimize the squared differences between actual function values and predicted values. if we denote by e_p the quantity:

$$e_p = y_p - o_p \tag{3.3}$$

We can determinate the normalized mean sum of squared error (SSE) as:

$$SSE_p = \sum_{p=1}^{n} (y_p - o_p)^2 \tag{3.4}$$

To minimize SSE/2, like in any steepest-descent method, we use backprobagation to compute the gradient of the loss with respect to w. If $SSE/2$ is minimized, also SSE will be minimized too. Finally, we can update the values of w and x as:

$$w(i,h) \leftarrow w(i,h) + \mu \sum_{p=1}^{n} (y_p - o_p)x(h)v_p(h)(1 - v_p(h))u(i) \tag{3.5}$$

$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^{n} (y_p - o_p) v_p(h) \tag{3.6}$$

Where μ is a scalar value called **learning rate**. This value is one of the most important in all the NN. The hyperparameter μ can heavily influence convergence and the speed of our network. This process is repeated a number of times which is called **Epochs** and this parameter is also chosen by the user. Epoch means one pass over the entire dataset [27] and If this value is too large, the model can have the so-called **overfitting** problem. Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize [28]. To avoid this problem it's possible to choose a tolerance: in this way the algorithm only stops when the difference:

$$SSE_{new} - SSE_{old} \leq tolerance \tag{3.7}$$

Before starting to deal with an NN, it's important to emphasize that the values of w and x must be chosen in a smart way. A recommended way to initialize that, is to choose values from a uniform distribution $U(0, 0.1)$ or using Xavier [29] initialization.

At this point, everything is almost ready: since the number of input and output nodes are fixed, the only problem is to decide a 'correct' number of hidden nodes and a suitable value for μ. Starting with choosing hidden nodes, there are two problems:

- As NN use a steepest-descent algorithm, a well-known problem arises: they can only guarantee convergence to a local minimum. If there is more than one local minimum, this could means that the solution found is not the best possible.
- The use of a large number of hidden nodes can avoid the problem of multiple optimal, but this means a huge computational effort to compute both w and x and can cause an overfitting problem.

Therefore, a balance between the number of hidden nodes and the performance of the network must be found. The only possibility is to make attempts, and see the results. The last hyperparameter to choose is the value of μ. This is a crucial parameter, as said before, because it can generate other well-known problems called **gradient explosion** and **gradient vanishing**. The former phenomenon produce large values of $\hat{v}$ and this means that the weights w and x are very close to 1. This

implies that for wide values of weights the network lose its discriminatory power. The latter can dramatically slow down the learning process of the network. If the gradient assumes too low values, the weights stop changing and the loss value stops decreasing. Based on [10] we can update the value of μ at the each iteration in the following way:

$$\mu = \frac{log(m)}{m} \tag{3.8}$$

where m is the number of iterations. This is a very nice sequence which has some well-known conditions as:

$$\sum_{m} \mu_m = \infty \tag{3.9}$$

$$\sum_{m} (\mu_m)^2 < \infty \tag{3.10}$$

Now, the problem is that this sequence, at the beginning, has too many high numbers and the can NN diverge immediately. Numerous different solutions can be implemented, but one of best one is to keep the μ fixed to a very small value. Last but not least, it's important to normalize or standardize the input values. Normalization techniques are essential for accelerating the training and improving the generalization of deep neural networks [30]. If we skip this step, the scale of the input values will cause a w and a x very close to one, as in the case previously mentioned. This fact would imply that the network will lose all its discriminatory power.

3.3 The Transformer architecture

3.3.1 Introduction

The transformer is a neural network architecture introduced by Vaswani et al. [23] in 2017. It was devised for natural language processing (NLP) application, but nowadays it is used in different fields such as computer vision and biological sequence analysis. Vaswani's goal was to find an architecture that:

- does not use a fixed size vector for the state (as a recurrent neural network (RNN)[31])
- does not need a lot of layers and parameters to learn long-distance dependencies (as convolutional neural network (CNN) [32])

- has an explicit way to model hierarchy (fundamental for NLP task)
- it is simple and consequently can be easily parallelizable

The result is a very straightforward method based on "self-attention", a multi-layer perceptron (MLP) and some additional tricks like positional encoding to make everything converge. The transformer, thanks to this attractive design, has introduced almost no inductive bias with an even greater generalization power than an MLP.

3.3.2 Multi-head attention

Most Machine learning tools are inspired by the way the human brain works. MLP neurons function like real neurons as they are connected, they share a weight, we have an activation function and so on. Even in the attention case, this mechanism works in a similar way to human attention. Imagine you want to take a break from reading this thesis and have these three possibilities: have a coffee, look at your Instagram profile or smoke a cigarette. The first type of attention is nonvolitional and depends on the salience of the objects around you. For example, if you have an Instagram notification, attention is unintentionally directed to the smartphone. After replying to the new post by "Chiara Ferragni" you finally want to have a coffee. This time you are no longer affected by the salience of objects and your attention goes directly to the coffee machine. So, as we understand from this example, we have two types of attention: a voluntary one that depends on our needs and an involuntary one that depends on the salience of objects. In machine learning, voluntary signals are called queries. Given any query, we select the right sensory inputs (for example, intermediate feature representations) via the attention mechanism. These sensory inputs are called values in the context of attention mechanisms. More generally, each value is associated with a key, which can be thought of as the non-voluntary signal of that sensory input [22]. Vaswani et al. uses a variant of this attention called "self-attention" which seeks speed and simplicity. This formulation takes this particular name because the attention is focused on the input itself, i.e. the inputs can interact with each other (i.e. calculate the attention of all the other inputs with respect to another). In this way we have a direct flow of information for each input. This is the main strength over CNN and Long short-term memory (LSTM). At this point, to understand how this method works, we introduce an input vector $I = (i_1, i_2, ..., i_{n-1}, i_n)$. Now, if we want to re-represent i_2 3.2, using this mechanism, first we linearly transform it into Q_2. Then we linearly transform every position of the input into keys: K_1, K_2, ..., K_n and values V_1, V_2, ..., V_n. These transformation steps can be thought of as a way to extract features from the input and, for this reason, attention layers can be seen as feature detectors. Through this transformation we are projecting our data into

a new feature space where the dot product is a good measure of similarity. After multiplying the Q_2 query by each key,we can finally apply a softmax function. The last step in achieving the embedding of i_2 is to compute a convex combination of the weighted values from the results of the softmax function. Then the result is passed through a multilayer perceptron. The final formula for computing attention is as follows:

$$Attention = softmax(\frac{Q\ K^T}{\sqrt{d_k}})V \tag{3.11}$$

where $\sqrt{d_k}$ is a scale factor to prevent the computation from blowing up and causing numerical instability. This is also very fast and efficient to compute on a GPU since it consists only of two matrix multiplications. In a single self-attention layer we have all the data available, thanks to its structure, but it is mixed since it is a convex combination. The problem is that it is not possible to select different pieces of information from different places and evaluate them by themselves from the other input. The solution is to use multiple attention layers in a solution called multi-head attention layers. In this way, each head can consider different aspects of the same problem. All this work can be easily parallelized and can be done by reducing the size of the latent space. It is unclear whether this mechanism has better expressiveness than an LSTM. In fact, from the general Machine Learning theorem, an LSTM can adapt to any possible function [24], so why can the transformer perform so well? These results may be due to the fact that this method works very well with stochastic gradient descent (SGD) since the dynamics and attention gradients are really simple functions.

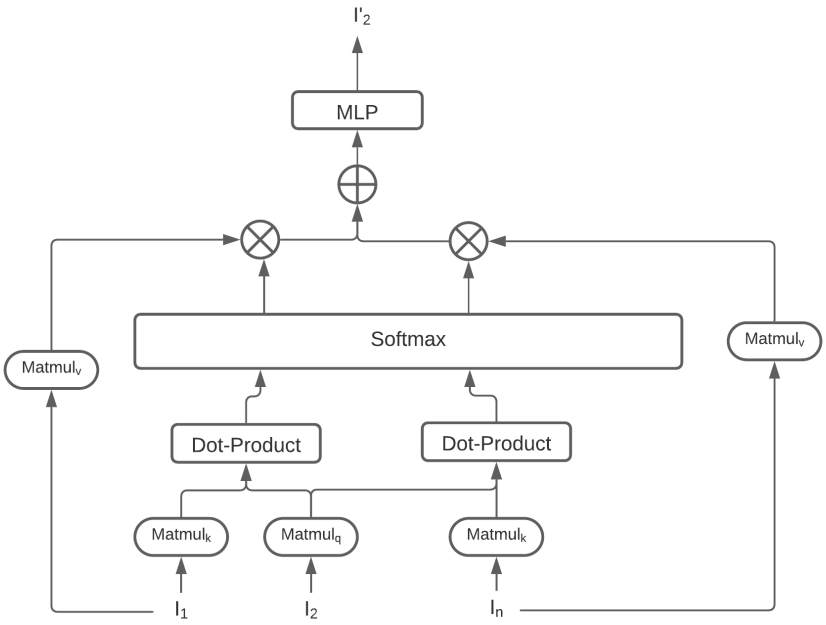


Figure 3.2: Self-attention mechanism.

As you can see from Table (3.4), attention is very attractive when the length of your input sequence (n) is lower than the size of your embedding space (d). Moreover, as we have already said, the transformer does not compute sequential operations to be highly parallelizable. On the other side, CNN and LSTM, on the other hand, are quadratic with respect to the new representation space (with size d). CNN also has another term k which represents kernel size. In general, this value can be neglected as it is usually much lower than k and d.

Per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions [23].

Layer Type	*Complexity per Layer*	*Sequential operations*
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(\log_k(n))$

3.3.3 Positional Encoding

The approach proposed by Vaswani is designed to be fast and parallelizable therefore, for this reason, sequential operations are not allowed. Despite LSTM, in the Transformer, we have no idea of the order of the input. In other words, transformers have permutation invariant inductive bias. This means that regardless of the input order, the final result is always the same, and if we think about the architecture of the transformer this is quite obvious. We will come back on this point later in section 3.4 Since it is a weighted sum, if we change the order of the addends the final result is always the same. This could be a problem, especially for a task where the order of input and output is crucial. The solution is to inject some information about the positive or relative positions of the input tokens.

Absolute Position Encoding

To make the model understand the sequence order of the input, we can inject a positional encoding into the representation of the input. This encoding can be learned (using the normal gradient descent method) or it can be fixed from the beginning. Vaswani et al. [23] show that for NLP problems the final solution is very similar for both solutions. In their work, Vaswani et al. use sine and cosine functions at different frequencies:

$$\mathrm{PE}_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{3.12}$$

$$\mathrm{PE}_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{3.13}$$

where pos is the position and i is the embedding dimension.This layer is introduced immediately after embedding the input vector features, as can be seen in Figure Figure 3.3.

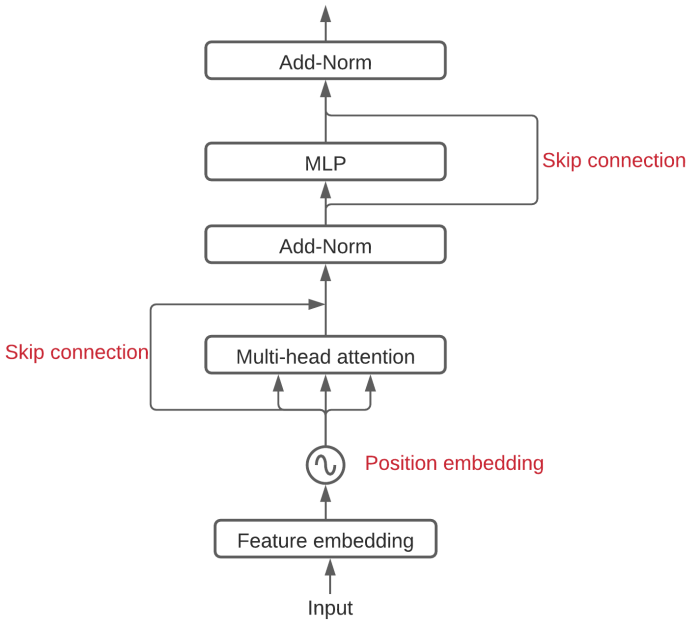


Figure 3.3: Transformer Block.

The question now is: is this sine-cosine embedding necessary? Why don't we assign an increasing number to each time step? This seems like a very basic but valid idea: to the first input i_1, we assign "1" while to the second input i_2, we assign "2" and so on. With this solution, there are two main drawbacks:

1. when we deal with long sequences, the input value can become extremely large.

2. the model may not see any sample with one specific length, which would damage the generalization of our model.

A more sophisticated idea is to assign a value in the interval $[0, 1]$. In this way, the first input will always have the label "0" and the last one the label "1".The above problems, with this solution, are solved but this idea introduces a new criticism. If the length of the input changes during training, the time step does not have a consistent meaning between different inputs. If position embedding may seem like an easy task now, let's begin to understand why the authors opt for this sine-cosine solution. First, we do not consider a single number as "0" or "1" but a d_{model}-dimensional vector for each position constituted of sine and cosine. To understand how this embedding works, we can consider the following example 3.4:

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	10 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1

Figure 3.4: Binary counter: All the number in the range [0-15] expressed in a binary format[33].

In this type of algebra, the rate of change is fixed. The lowest significant bit (LSB) changes with each step while the second LSB changes every two steps and so on. The sinusoidal counterpart mimics this behavior but continuously (Figure 3.5).

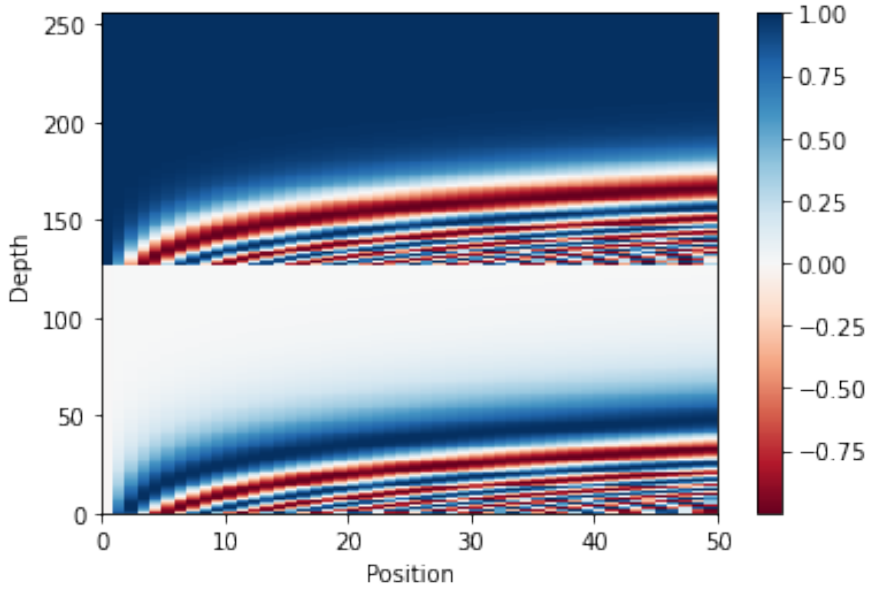


Figure 3.5: The 256-dimensional positional encoding for a sentence with input length set to 50.

Position information may seem non-crucial to the transformer architecture. Reading the article, the main question is: can this "noise" affect the behavior of this deep and complex model? To study how position encoding affects the attention mechanism, we printed the attention activation of the last Multi-head layer. In

Figure 3.6, we show the transformer without the skip connections, while in Figure 3.7, we report the result of a "usual" transformer block. By removing the skip connections, subsequent levels lose all knowledge of the order of the input sequence. In fact, the chart shows horizontal rather than diagonal pattern lines.

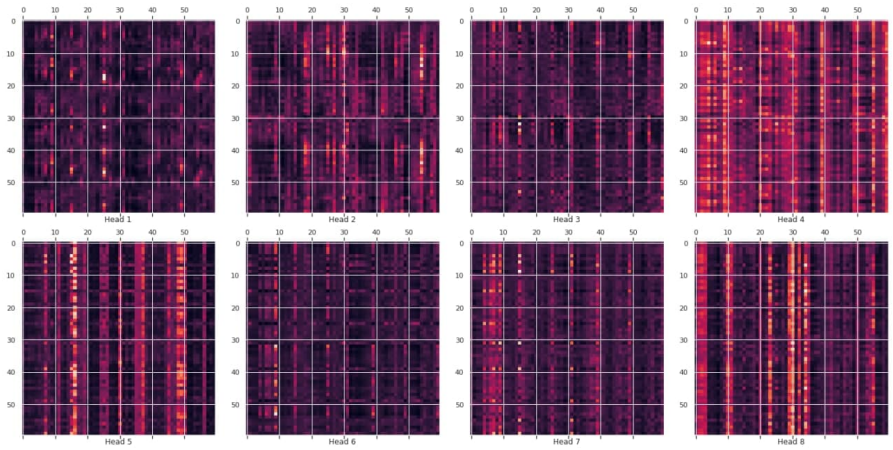


Figure 3.6: Attention without skip connections.

On the other hand, when subsequent layers have a notion of location, attention maps show a strong diagonal focus. This implies that the position encoding trick really helps the transformer understand the order of the input sequence. Furthermore, in this architecture, residual connections are used not only to mitigate the vanishing gradient problem but to carry information to subsequent layers as well. The same result can be achieved by replacing the skip connections with other position-encoding layers.

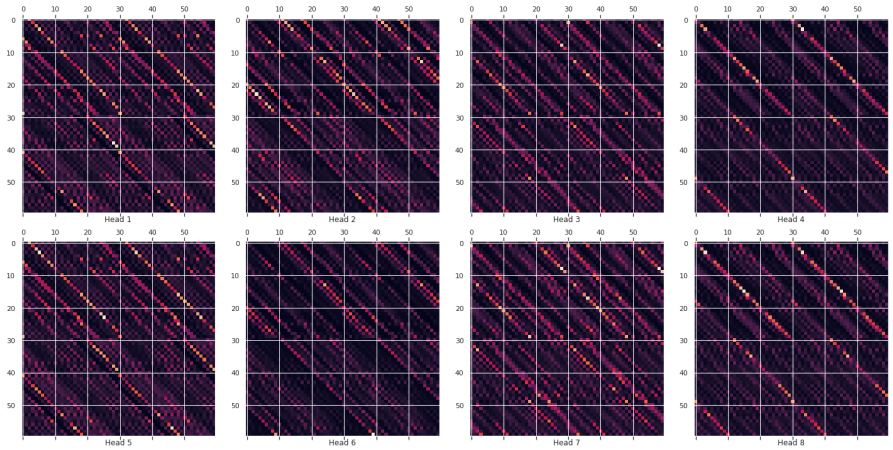


Figure 3.7: Attention with skip connections.

3.3.4 Relative Position Encoding

Let us consider the question "Marco, where are you working on your thesis?

- I'm writing my thesis in my childhood house at latitude: 43.5593691 and longitude: 11.5361732.
- I'm writing my thesis in my childhood house in a little town located 40 km south-est from "Firenze".

To answer this question both answers can be considered correct. In the second case we describe our position in relation to another city. The questioner can easily understand where we are working only if he/she knows where Firenze is. In the first answer, everyone can quickly determine where we are without further information. we describe my position absolutely and no other contextual details are needed, while "40 km south-east of Firenze" describes a relative position. We have already seen how absolute positional encoding can be embedded into the transformer, but we can still examine an extension of self-attention to consider the pairwise relationships between input elements: namely, a Relative Position Encoding [34]. Starting from the original work of Vaswani, each attention head operates on the same input $I = (I_1, I_2, ..., I_n)$ to produce a new representation $Z = (z_1, z_2, ..., z_n)$ with the same shape. So, each element of the vector Z can be computed as:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(I_j\, W^V) \tag{3.14}$$

Where W^Q, W^K and W^V are parameter matrices. These parameter matrices are unique per layer and attention head. While α_{ij} represents the weight coefficient and can be written as the output of a softmax function:

$$\alpha_{ij} = \frac{exp\, e_{ij}}{\sum_k^n exp e_{ik}} \tag{3.15}$$

while e_{ij} is simply the output of the proximity function:

$$e_{ij} = \frac{(I_i\, W^Q)(I_j\, W^K)^T}{\sqrt{d_{model}}} \tag{3.16}$$

The work of Shaw et al. [34] proposed some variations to these formulas in order to incorporate relative information between the input. As shown in Figure 3.8, each node is linked to k-neighbors where $k < n$. The magnitude of each link describes the strength of the connection between I_k and I_j. The edge between I_i and I_j is represented by the two vectors a_{ij}^V and a_{ij}^K. These terms are shared among the different heads of the attention mechanism. At this point, by modifying the

previous formulas it is possible to propagate the edge information to the output sublayer. And consequently:

$$e_{ij} = \frac{(I_i\, W^Q)(I_j\, W^K + \alpha_{ij}^K)^T}{\sqrt{d_{model}}}$$

and therefore:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(I_j\, W^V + a_{ij}^V)$$

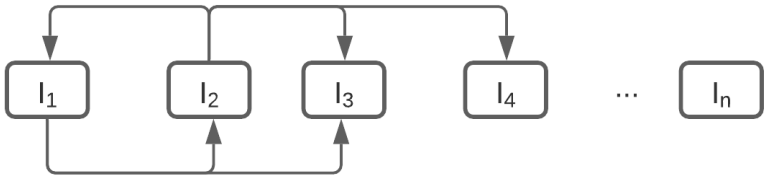


Figure 3.8: Relative encoding graph representation for node I_1 and I_2. K is set to 2.

3.3.5 Encoder-Decoder architecture

The original transformer uses an encoder-decoder structure, consisting of stacked layers of encoder and decoder (Figure 3.9). The encoder layers consist of two sublayers: self-attention followed by a position-wise feed-forward layer. The decoder layers consist of three sublayers: self-attention followed by the encoder-decoder attention, followed by a position-wide feed-forward layer. It uses residual connections around each of the sublayers, followed by normalization layer. The decoder uses masking in its self-attention to prevent a given output position from incorporating information about future output locations during training.

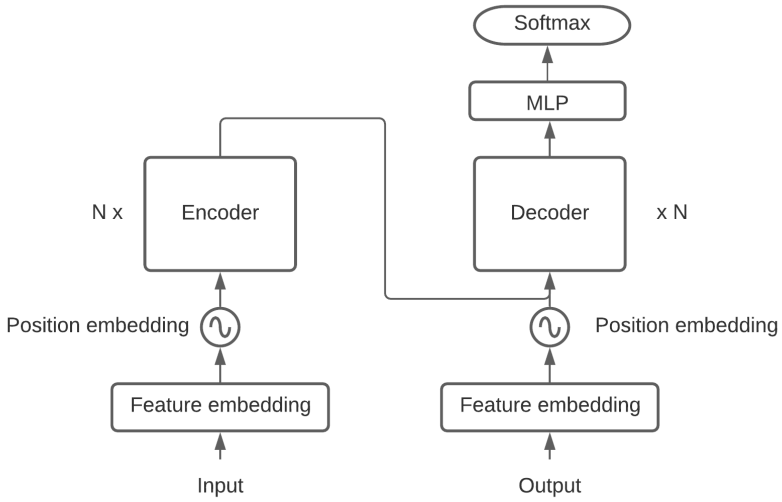


Figure 3.9: Transformer encoder-decoder architecture.

3.3.6 Parisotto Variants

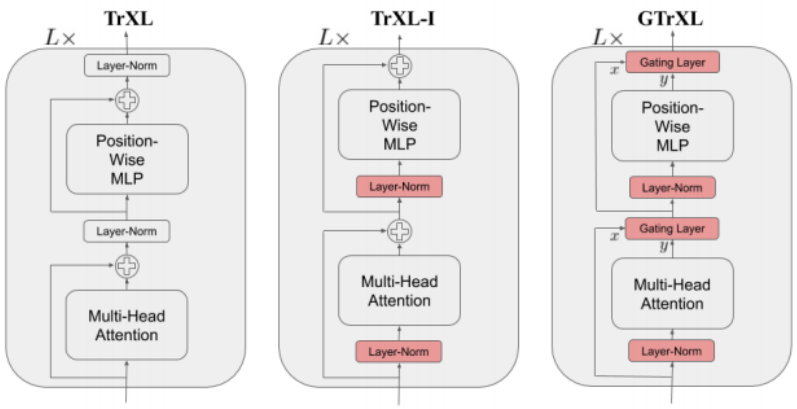


Figure 3.10: Transformer variants, showing just a single layer block [35].

Parisotto et al. [35] want to experiment the Transformer model in different RL settings to check if the architecture proposed by Vaswani can be applied to these kinds of problems. Parisotto uses the attention mechanism exploiting only the encoder part. The idea is to use a transformer encoder to encode the value of the state. As shown in Figure 3.10 the authors try two variants of the standard

transformer: the TRXL-I and the GTRXL. The main changes in the TRXL-I are inspired by the work done by [36], [37] and [38]. In these works, they show as the "Identity Map Reordering" helps the network to converge faster and better to the final result. This name is introduced since we have an identity map from the input that flows directly to the last layer. This is in contrast with what happens in the TRXL, where the input values go through two different normalization layers. To avoid this behaviour, the position of the normalization layers are changed as shown in the central representation of figure 4.8. This variant should help the agent to learn reactive behaviours before memory-based ones. This solution is particularly useful in certain types of environments especially the ones where the agent needs to learn how to perform an action before it can learn how to remember that particular action. Parisotto's work does not stop here and proposes an additional gating mechanism to add to the TRXL-I variant. The result is the GTRXL model. To enhance performance and optimization stability Parisotto removes the "add" operations with a gating layer. In their paper, several implementations of this layer are proposed but, the one that they find most successful is based on the following GRU type mechanism:

$$r = \sigma(U_r\, x + W_r\, y) \tag{3.17}$$

$$z = \sigma(U_z\, x + W_z\, y - b_f) \tag{3.18}$$

$$h = tanh(U_f(r \odot x) + W_f\, y) \tag{3.19}$$

$$f(x, y) = z \odot h + (1 - z) \odot x \tag{3.20}$$

where σ is the activation function, $\odot$ is element-wise multiplication and x and y are respectively the result of the skip connection and the input of the previous layer. To enable a much larger contextual horizon than would otherwise be possible, Parisotto uses the relative position encodings described in Sec. 3.3.4 and memory scheme used in Dai et al. [39].

3.4 Inductive biases

Inductive biases are a set of assumptions that a learner uses to make predictions about a new sample unseen in the training phase. The only way to construct an unbiased learner is to have a training set that coincides with the test set. Namely, the learner should have seen every possible sample before making a prediction. This is unfeasible, unrealistic and even useless. For this reason, the learner has to make

some hypotheses about the nature of the target function. These assumptions are called inductive biases and allow mapping the objective function with a reasonable number of samples and retrieve the correct solution in fewer iterations of the algorithm. Depending on the problem we have to face, the right tool with the most appropriate bias must be selected. Select the wrong algorithm that introduces unfitted assumptions can adversely affect the performance of the learner and lead to a suboptimal solution. These biases are present in every tool, algorithm and regularization technique used in machine learning.

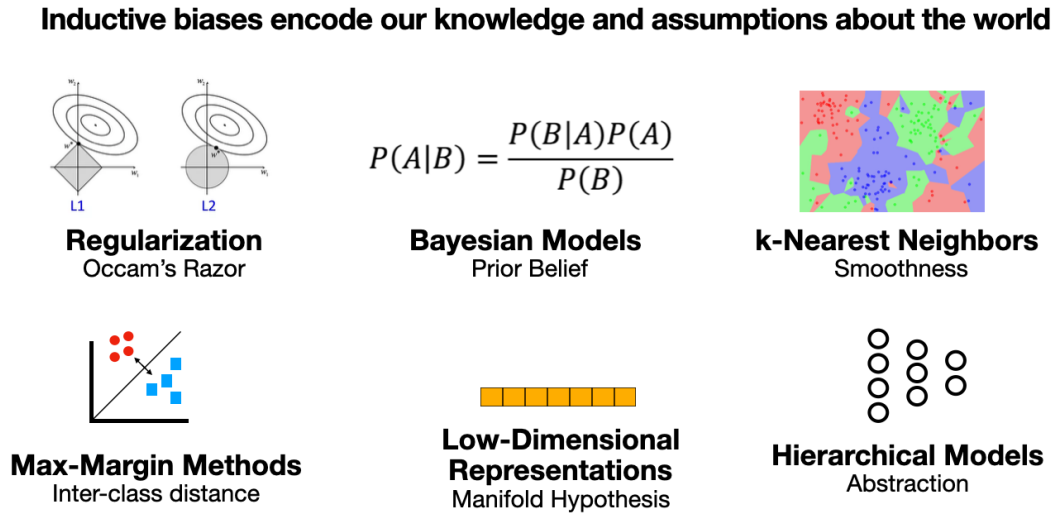


Figure 3.11: Inductive biases examples [40].

Based on Figure 3.11:

- A clear example comes from the Euclidean (also known as the L2) norm. In general, ensemble models perform and generalize better than standard leaner, for instance, a random forest compared to a decision tree. For this reason, with the L2 norm, we prefer to share weights among all the nodes instead of having few nodes with all the decision power. In general, this is not true. There are some empirical evidences of this fact, but it can no work in some circumstances. As already said before, introducing a bias in the networks is based on our beliefs on the nature of the problem.
- This fact is even more evident when we look at some Bayesian models. In that learner, if we already know how to model the final function, we can add an informative prior knowledge. This solution helps the learner to find earlier a solution and requires less data to obtain the final result. Knowing the final shape of the distribution implies that the learner has to establish only the

expected value and the standard deviation. Obviously, if the assumptions on the informative prior were wrong, the model will need more data and iteration to find the optimal solution.

- The K-nearest neighbours model assumes that samples that are neighbourhood in feature space belong to the same class. So, given a new record with an unknown label, it will assign the new class based on a majority vote in its immediate neighbourhood. In general, again, this fact can not be true. We can have samples that share different properties but belong to distinct classes. We will come back to this example in a while.
- Then there are max-margin methods that aim to draw boundaries between classes, maximizing the width of that specific boundaries. If we have a large margin between samples belonging to different labels should generalize better than an arbitrary boundary one. This assumption is the one used by support-vector-machine (SVM). With this type of bias, we are sure to retrieve the best possible boundaries, but it's a quite complex optimization problem that does not scale with an increasing number of data and dimension space.
- Minimum features bias: unless there is good evidence that a predictor is useful, it should be deleted. This assumption is the one behind feature selection algorithms such as PCA, SVD, TSNE and so on.

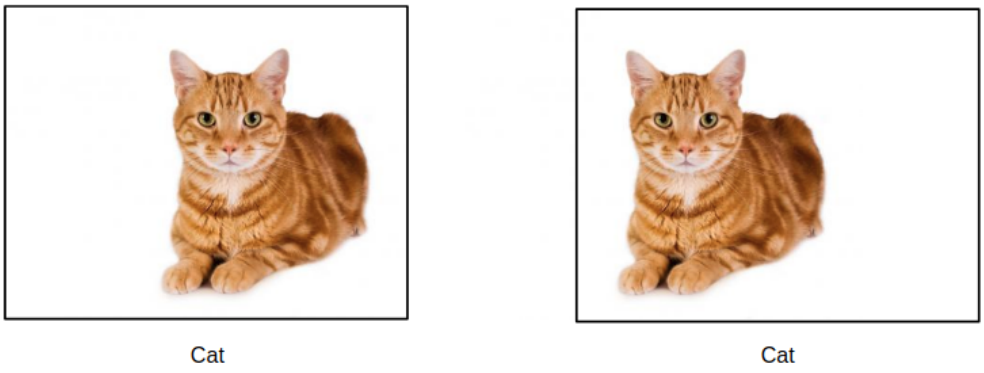


Figure 3.12: Same cat different position in the image.

Let us consider Figure 3.12 which represents two images of the same cat but translated. If we train a classifier to distinguish different animals, we want that these two images will belong to the same class. If we use a K-nearest neighbours approach based on the images' colour distribution, what we obtain is depicted in Figure 3.13.

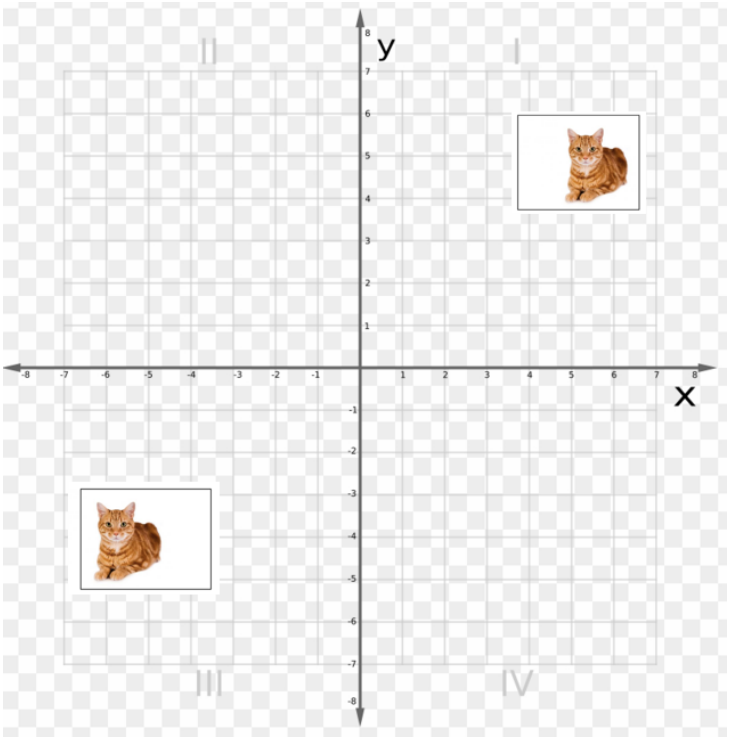


Figure 3.13: Representation of the two cats images of Figure 3.13 in the features space.The two cats are far away in the new feature space. For a classifier is difficult to predict the same class for both.

They are very far away in this feature space so, it is very likely that they will be classified as different objects even if it is clear that they are equal. The inductive bias introduced this time does not help to retrieve an optimal solution due to a wrong starting assumption.

On the other hand, a convolutional neural network can easily solve this task thanks to its translation equivariant property. To show what this means, let's consider a pooling layer [41]. The pooling layer and convolutional layer works differently but, for the sake of simplicity, we will examine only the former. These considerations can be easily extended to the convolutional layers.

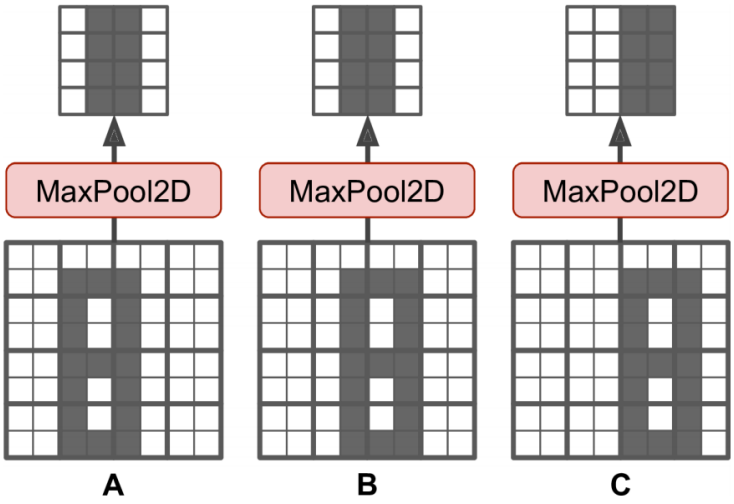


Figure 3.14: Pooling layer invariant to small variation [42].

As displayed in Figure 3.14, the outputs of the max-pooling layer for images A and B are identical. This is what translation invariance means: different inputs same results. For image C, the output is different: it is shifted one pixel to the right. This means that the pooling layer is not translation invariant but equivariant. Namely, we obtain the same transformation but, it's non located in the same place. The new location can be computed deterministically. So, what we have learnt from this basic example is that we need to choose the model that introduces the right biases to solve a specific problem. If we do so, we can obtain better results by being more sample efficient and more aware of what the different layers are performing.

To understand in detail this data efficiency point, we will introduce a new example. We know that CNNs are translation but not rotation invariant. If we rotate the image, the CNN is not able anymore to correctly predict the correct label. This fact is particularly obvious if we look at the activation map in Figure 3.15.

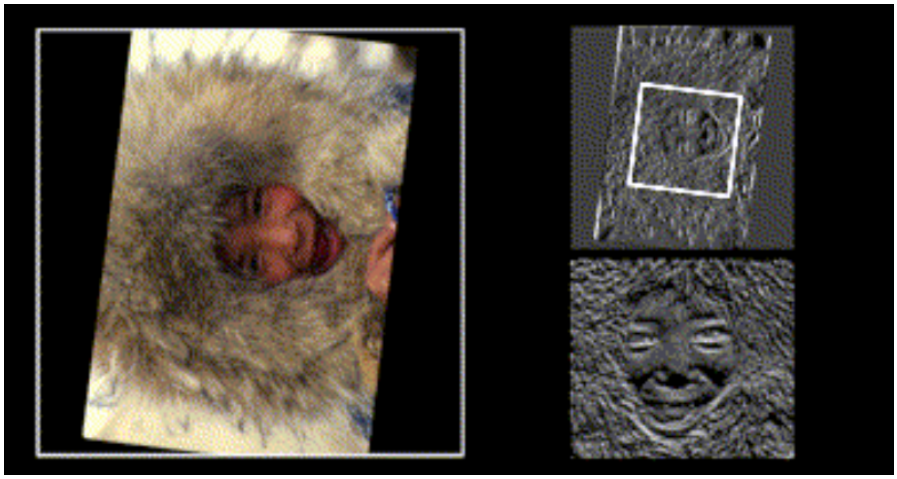


Figure 3.15: Convolutional layers are not translation invariant/equivariant [43].

For the same rotated input, the activation result is always completely different. To mitigate this problem, we can use a very common technique known as data augmentation. For every possible image, we rotate it by a random angle and we use it as additional data for our training phase.

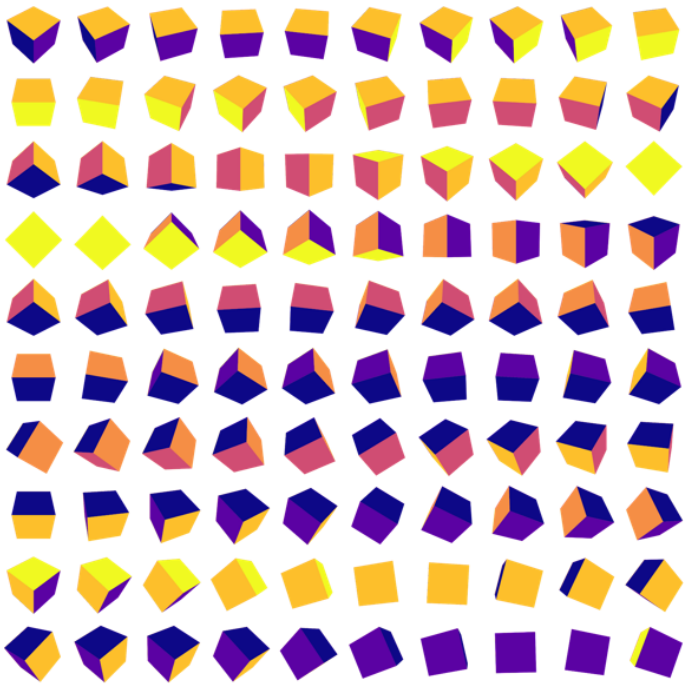


Figure 3.16: Data augmentation brute-force approach [25].

Data augmentation is a brute-force approach to teach a model how to "emulate" symmetry-awareness. For 3D data, data augmentation is expensive ~ 500 fold augmentation, Figure 3.16 [44]. This solution can be very computationally expensive, and the network has to learn the same specialized filters to understand that these images are all the same. This reduces the discriminatory power of the network. Another solution is to create a CNN that is not only translation but also rotation invariant. Several solutions can be applied, like: rotating the action map, rotating the filter or using some specific Harmonic layer [43].

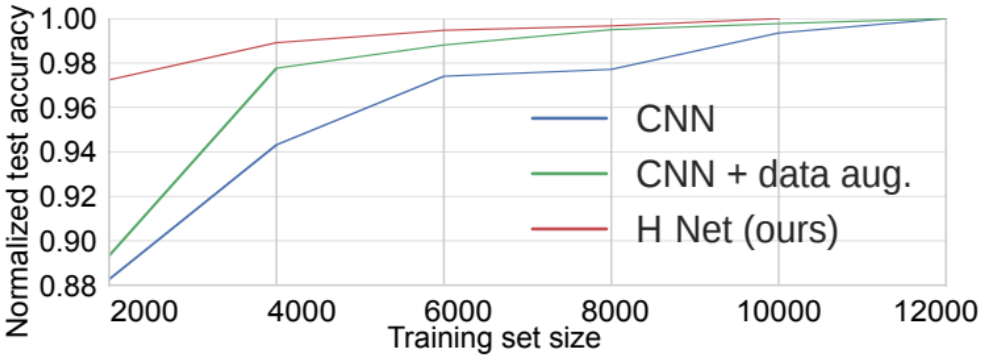


Figure 3.17: On the rotated MNIST dataset, H-Nets are more data-efficient than regular CNNs, which need more data to discover rotational equivariance unaided. [43].

The benefits of this biased introduction are that all neurons in the network can be used to learn important feature and no additional data is required. On the MNIST dataset [45], Figure 3.17, the harmonic network (Hnet) with only 2000 samples is able to overcome 0.98 of accuracy [43], while the usual CNN is not able to overcome 0.9 using the same amount of sample. Moreover, it requires more than 10000 inputs images to be able to match the performance of Hnet. Obviously, this network performs especially well in that kind of task. Hnet on a normal task like imagenet [46] struggles to perform better than usual CNN. Again, depending on the type of problem we are facing, we make different sets of assumptions and we use different layers/architectures.

The obvious question at this point is, “which bias is best?”. The no free lunch theorem of machine learning [47] shows that there is no one best bias and that whenever algorithm A outperforms algorithm B in one set of problems, there is an equally large set of problems in which algorithm B outperforms algorithm A. For this reason, we think that for our thesis, understand which bias to introduce and which bias to avoid is crucial. We have a limited amount of data, so we have to find the more appropriate architecture that will extract all the possible values

from this data. What makes this task even more challenging is that we have to use an RL agent. Training this type of algorithm usually, need a large amount of data to avoid to stuck in some local optima solution.

Inductive biases per network types.

Network types	*Inductive bias*
RNN	Equivariance over time
RNN	Recency bias
CNN	Translation bias
CNN	Locality bias
Graph Neural networks	Equivariance over entities and relations bias
Soft Attention	Equivariance over permutation
Deep architectures	Complex functions = composition of simpler ones

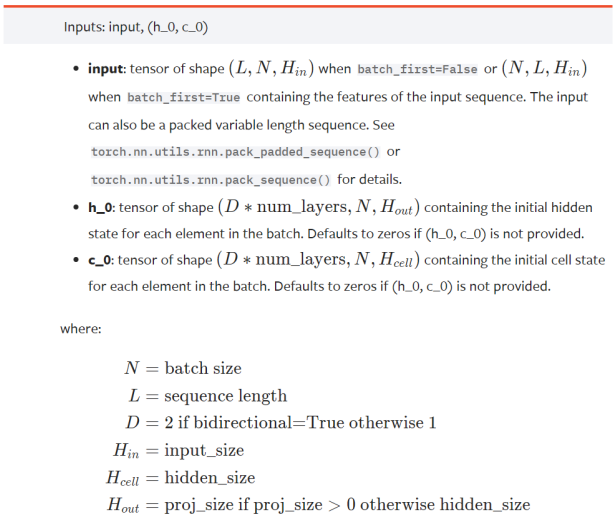
Let's start with the LSTM or, more in general, RNN architecture before introducing the Transformer. The most common bias in RNN is the time-invariant bias. In general, this is a desirable solution to incorporate into a model. If an event appears more than one time the learner is able to recognize it and produce the same kind of output. This is a suitable property for time series since we have seasonality and periodic trends that are important to spot. Another bias that affects RNN is called recency bias. Recency bias is a cognitive bias that favours recent events over historic ones and gives greater importance to the most recent event. For instance, if an asset price has dropped for 30 days but in the last few days is starting to grow again, the model will tend to predict an increasing value. This bias is due to the architecture of RNN. This model, due to the hidden state, is not able to learn long dependencies but tends to focus only on short term correlations. This type of bias reflects the trends of the asset and can help to reduce the training data needed to solve this problem.

Let's focus now on CNN. We have already talked about translation-invariant so try to focus now on Locality bias. In CNN, we have a "local receptive field connectivity". In an image, the correlations between pixels are very high when they are near and become weaker when they start to be far away from each other. For this reason, CNNs have filters that are very small compared to the input photo. This means that the first CNNs' filters will focus only on very particular details of the image. Stacking more CNN, deeper layers will have a wider receptive field that will develop more high-level features. In this way, we end up creating a sort of hierarchical structure. As you see from the table 3.4, Deep architectures introduce a new bias. Even if it is not theoretically clear what is the additional power gained

by the deeper architecture, it was observed empirically that deep RNNs, CNNs and transformers work better than shallower ones on some tasks.

Finally, let's talk about self-attention and transformers. Transformer's inductive bias is more relaxed than either recurrent or convolutional architectures. This could seem a great result with respect to CNN and RNN but not exactly. In fact, due to the absence of relevant biases, this model is very data-hungry. It takes that much data to catch up to the hardwired priors of RNNs/CNNs, and then go on to exceed them with better learning. With more data, transformers exceeded RNNs for almost everything and seem to exceed even CNNs for image classification, as stated for example in: "Attention Augmented Convolutional Networks", [48], which looks at CNNs, CNNs+attention, and pure-attention for image classification, where the pure-attention NNs are almost catching up to the CNN. The only evident bias in Transformer is the equivariance over permutations. This means that for the same input it is possible to permute the features but the final result does not change. This is clear if you think about how attention is computed. This is a good property to have but not so relevant.

Exploiting their biases, LSTM and Transformer are the best candidates to solve this task. The former can spot seasonability and but it's not able to learn long term relationships. LSTM based their actions focusing especially on the last steps. While Transformers can learn both long and short term patterns, the amount of data required to be trained increased significantly. Even if LSTM, seems a bit better than Transformer to face this particular task, we will not be able to carry any type of test with this architecture. As already said reported in chapter 1 the input data will be a tensor of length four (batch_size, number of assets, number of days, feature size). Looking at the Pytorch [49] (the machine learning tool that we used to designed our models) documentation in Figure 3.18 the required shape is 3-dimensional. To make this model work, we should have compacted the number of assets with the number of days. This solution obviously does not make any sense, so for this reason, we prefer to adopt the transformer to work with this type of data.



Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the batch. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the batch. Defaults to zeros if (h_0, c_0) is not provided.

where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{cell} &= \text{hidden\_size} \\
H_{out} &= \text{proj\_size if proj\_size} > 0 \text{ otherwise hidden\_size}
\end{aligned}
$$

Figure 3.18: LSTM from pytorch documentation [50].

Chapter 4

Reviews and Proposals

4.1 Old paradigm

4.1.1 Introduction

Using reinforcement learning and deep architecture, like a transformer, to approach Portfolio optimization is a very attractive scenario. In the literature, we can find different types of solutions that use these tools. Starting from the work done by Xu [51] and proceeding with the one of Kim et al. [52] we have very encouraging results using these types of tools. In both papers, their approaches overcome different baselines, both shallow and deep. In fact, we have methods like the OLMAR (On-Line Portfolio Selection with Moving Average Reversion) [53] and UCRP (Universal Portfolios Review and Online Convex Optimization) [54] which follows a more standard and mathematical approach but also ADDPG [**xie2021**] which use a reinforcement learning agent. In particular, the work done by Kim et al. looks very promising. The model is trained and evaluated with assets of nine Dow Jones companies representing each sector - industrials (MMM), financials (JPM), consumer services (PG), technology (AAPL), health care (UNH), consumer goods (WMT), oil gas (XOM), basic materials (DD) and telecommunications (VZ). The OHLC prices and trading volume data of the assets are collected from Yahoo Finance. The data for 18 years from 2000 to 2017 are used for training, and the data for the period from 2018 to April 2020 is used for evaluation [52]. To overcome the aforementioned baselines a new model called DPGRGT (Deterministic Policy Gradient with 2D Relative-attentional Gated Transformer) is introduced. As the name said, Kim uses the DDPG algorithm as a reinforcement learning agent and the variation of the standard transformer ideated by Parisotto et al. [55].To apply this algorithm to real case scenarios, the authors proposed realistic constraints of transaction costs. These constraints have a significantly negative impact on portfolio profitability and must be considered to obtain a down-to-earth

result. While many authors and papers focus on predicting future prices and built portfolios based on the prediction, Kim et al. design an RL setting in which the state-encoder takes into account the time series of different assets at the same time. In this way, the new method can learn not only the behaviour of the single asset but the correlations among the set of the input assets. This work is done by the transformer that can learn long-time relationships inside the portfolio's tools.

4.1.2 Problem Statement

Since a financial state is only partially observable, their study employs an observation set of historical prices and trading volumes up to time t to represent a state at time t [52]. An observation S_t is composed by what we have defined $OHLC$ plus trading volumes for each asset at time t, i.e.

$$s_t = (O_t, H_t, L_t, C_t, V_t) \tag{4.1}$$

Each feature is a t by $m+1$ matrix, where the rows represent the time axis and the columns represent the assets axis that consist of cash and m assets. For how this state is formulated, regardless of agent action, the following state s_{t+1} is always the same and equal to $s_{t+1} = (O_{t+1}, H_{t+1}, L_{t+1}, C_t, V_t)$. All the data is log differenced for the stationarity of the time series. Each state s_t is characterized by the last k days. So, a state is composed of the OHLC + Volume information from time $t-k$ to time t as it was a time series. We suppose that this formulation is introduced to replace the frame stacking technique used in many RL algorithms. Since we are working with partially observable MDP, the idea is to add information about the history. To understand why this solution is needed and helpful, let's introduce a naive but very understandable example. Suppose we aim to train an RL agent to play Pong. The state at each time t to be markovian should contain all the possible information to solve the problem. For instance, if we use a frame it is impossible to understand the ball direction, Figure 4.1. Instead, adding more frames of the last seconds of games, the agent can finally get the ball direction. The idea of Kim is similar to this one. If the agent knows the history of an asset can better choose the following action.

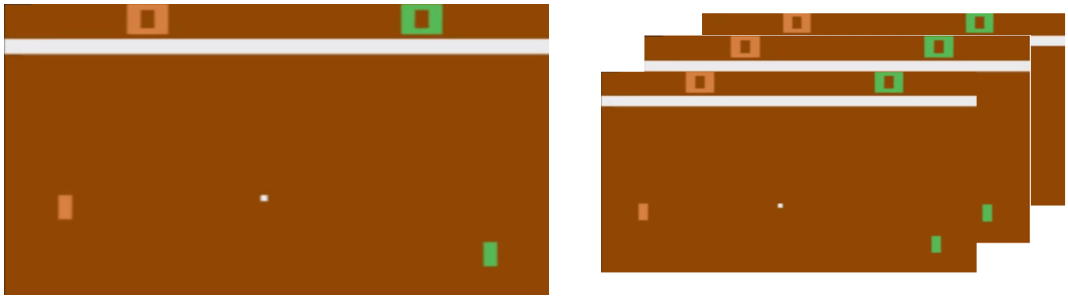


Figure 4.1: Pong game on Gym AI on the left. Stacking of frame method for RL on the right.

Instead, the action a_t is defined as the portfolio vector at time t, where the elements are the weights of resource allocation in cash and of m assets, and the sum of the weights totals one, i.e.

$$a_t = (a_t^0, a_t^1, ..., a_t^m) \quad \text{s.t.} \sum_i a_t^i = 1 \tag{4.2}$$

In their formulation, only long positions are allowed since $a_t^i \, \forall i$ must be always greater than or equal to zero. The way the action is defined is straightforward and helps a lot in the formulation of the reward. Using their words: "the reward is the risk-adjusted return for the action, with transaction costs applied" [52]. They define the previous portfolio value as the scalar value:

$$p_{t-1} = c_t * s_{t-1} \tag{4.3}$$

where c_t is the vector of the current closing prices and s_{t-1} is the vector of the previous shares. The new portfolio s_t is computed, giving the action a_t, using:

$$s_t = (p_{t-1} * a_t)//c_t \tag{4.4}$$

where // represents the integer division and returns the number of shares that will compose the portfolio at time t. At this point, they introduce two different types of transaction cost based on a conservative strategy. The transaction fee rate is set to 0.2% and the slippage rate is set at half of the proportional bid-ask spread. The bid price represents the maximum price that a buyer is willing to pay for a share of stock or other security. The ask price represents the minimum price that a seller is willing to take for that same security. A trade or transaction occurs when a buyer in the market is willing to pay the best offer available—or is willing to sell at the highest bid [56]. Since bid-ask data is not available, it's estimated based on

the work of Abdi et al. [57]. The proportional bid-ask spread d_t^i used for a single asset i at time t is computed as:

$$d_t^i = 2\sqrt{E[(\log(c_t^i) - \eta_t^i)\,(\log(c_t^i) - \eta_{t+1}^i)]} \tag{4.5}$$

where $\log(c_t^i)$ is the daily closing log-price at time t and η_t^i is the average of daily high and low log-prices at time t. The rebalancing cost for a single asset is the transaction fee and slippage proportional to the current closing price and change in shares of the asset [52]. Thus, the total rebalancing cost b_t is calculated as follows:

$$b_t = \sum_{i=1}^{m} c_t^i \left| s_t^i - s_{t-1}^i \right| (0.2 + 0.5 * d_t^i) \tag{4.6}$$

At this point we can compute how much cash we will have time t combining all these concepts:

$$s_t^0 = p_{t-1} - \sum_{i=1}^{m} c_t^i s_t^i - b_t \tag{4.7}$$

We have to underline that s_t^0 can become negative if the transaction costs b_t becomes really high. This possibility increase if action a_{t-1} is very different from action a_t. In general, this does not happens and when this problem occurs it's limited to a few hundred euros. This behaviour could be avoided if the value s_t^0 was computed before s_t.

At this point, we have a bit of confusion since they change the definition of the portfolio at time t. Based on (4.3) we had to write:

$$p_t = c_{t+1} * s_t \tag{4.8}$$

Obviously, it's impossible to provide information from the following time $t + 1$ so, they change the portfolio definition in the following way:

$$p_t = c_t * s_t \tag{4.9}$$

Finally they compute the log-return:

$$r_t = \log(p_t) - \log(p_{t-1}) \tag{4.10}$$

Since the return does not take into account the risk, the reward function used is the Sortino ratio (Eq. (1.9)). In this way, the RL algorithm will try not only to maximize the cumulative return but also to minimize the risk taken in every action. So, to evaluate the final model, two different metrics will be considered:

- Cumulative return
- Sortino Ratio

4.1.3 Model architecture

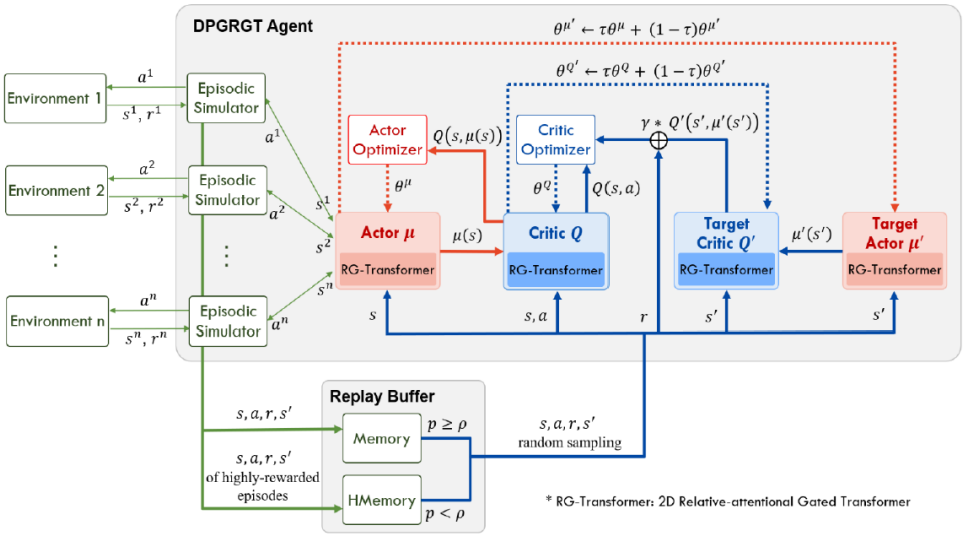


Figure 4.2: The overall architecture of DPGRGT model [52].

Figure 4.2 shows the overall structure implemented in the Kim et al. paper. This solution is sophisticated and uses different tricks to improve the sample efficiency and increase the performance using a sort of replay buffer. Starting from the input of the model they use multiple simulators to increase the experience of our agent. In fact, at time t, different actions are taken simultaneously and saved in the buffer. At first, we are quite sure that the actions will be various thanks to the "Uhlenbeck and Ornstein" noise [58], which is a temporally correlated noise. This solution, as in any other RL setting, is used to avoid ending in local optima. In this way, the agent has more information and can learn more from each state s_t. Unluckily has we will see see in the following chapters, taking a different action will not lead to a different s_{t+1}.

$$a = \mu(s|\theta^{\mu}) + dx_t \tag{4.11}$$

where $\mu(s|\theta^{\mu})$ is simply the output of the actor network, while dx_t is the Uhlenbeck and Ornstein" noise and can be rewritten as:

$$dx_t = \theta(\mu - x_{t-1})dt + \sigma\sqrt{(dt)}N(0,1) \tag{4.12}$$

where θ, μ and σ are parameter of the solution proposed in [58], while $N(0,1)$ is simply a normal distrubution with mean equal to zero and stantard deviation equal to one. Finally, to compute the new action a_t, Equation (4.12) is passed through

a softmax function. In general, this solution is the one used in most of the RL settings that implement the DDPG algorithm. The difference in this solution is the addition of the activation function. This approach is not wrong, but the parameters θ, μ and σ must be chosen carefully. In this case $\theta = 0.13$, $\mu = 0$ and $\sigma = 0.2$. We will come back to this point in the following chapters.

Another important change in this DPGRGT agent is the structure of the replay buffer. In this case, we have two different types of memory: one for all the types of episodes, and one only for the "cool" episodes, namely for the ones that renew the highest episodic reward. The agent sample the episode from the "cool" memory with probability ρ and from the other with probability $1 - \rho$. In this case ρ was set to 0.2. What it's not clear from the paper is if the episode saved in the later memory are also saved in the former. The introduction of this new cool memory should help the RL agent to learn from actions that were particularly good in certain circumstances.

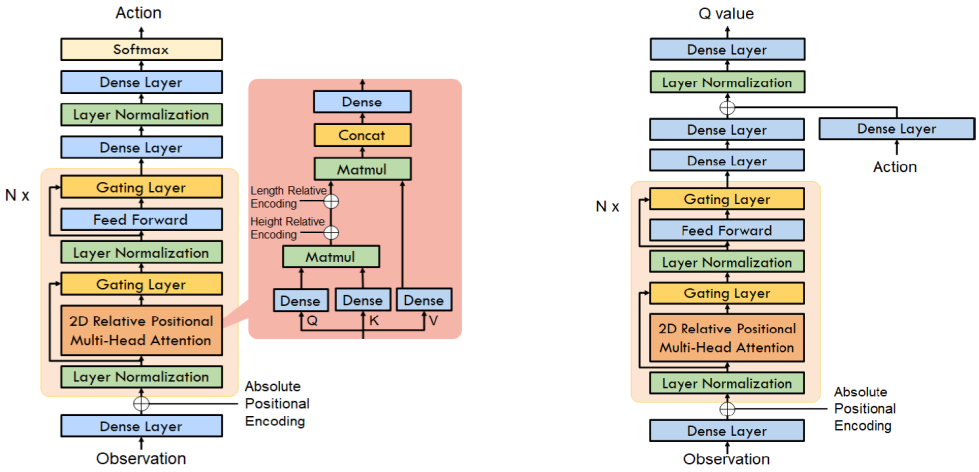


Figure 4.3: Schema of actor-critic architecture designed by Kim et al. [52].

The RL algorithm, as previously said, is a DDPG that is based on actor-critic architecture for continuous action space [59]. The target networks are updated using a soft updates strategy. A soft update strategy consists of slowly blending regular network weights with target network weights. Usually, we slowly mix the regular network weights and the target network weights but in this case the parameter τ is set to 0.85. This means that the update is quite fast since usual values are in range $0.95 \rightarrow 0.995$. The actor and critic share the same type of backbone, namely the 2D Relative-attentional Gated Transformer based on the idea of Parisotto et al. [55]. Since the standard transformer has shown limitations even on naive RL problems, the solution proposed by Kim is to use a variation

of the GRTXL proposed in Parisotto's paper. In fact, in this paper, we have two main changes to adapt the new version of the transformer to this particular and challenging problem. The first modification is introduced to solve the problem of the high dimensionality of the input data. Kim has a 4d input:

$$(batch\_size, number\_of\_assets, number\_of\_days, OHLC + Volume) \quad (4.13)$$

Based on the hyperparameters written in the paper the input has the following shape: $(batch, H, L, f) = (32, 10, 50, 5)$. In general, when dealing with LSTM/CNN2d/Transformer, they require 3d input data: following the Pytorch documentation the input and output tensors are provided as (batch, seq, feature) or (seq, batch, feature). In general, when we do Time series analysis, we proceed with a time series per time. Now, instead, we consider "number of assets" series per time and this implies an additional dimension. To overcome this problem, in the attention layer, the vector is reshaped to $(batch, H \cdot L, F)$. Thanks to this trick, the input dimension come back to the one used in a normal transformer. The pros and cons of this solution will be treated in the following chapters. The second main difference from the work of Parisotto stands in the number of layers used by the transformer. Parisotto's paper used a relatively deep architecture with 12 layers. They chose to train deep networks in order to demonstrate that their results do not necessarily sacrifice complexity for stability, i.e. they are not making transformers stable for RL simply by making them shallow [55]. The solution proposed by Kim opts for a three-layers architecture only. It's not clear if this is done to increase the stability of the networks, or for a too-large time/memory effort. The second hypothesis seems more realistic since no ablation study for the number of layers are reported and all code was run on Google Colab. Colab is a great tool to start working on ML stuff but, the non-pro version is very limited both from a memory and a computational point of view.

4.1.4 Results

The data of OHLC prices relative to the ten aforementioned stocks for 18 years from 2000 to 2017 are used for training, and the data for the period from 2018 to April 2020 is used for evaluation. The maximum length of a single episode is set at 50 days, and the initial investment at 100,000 USD. As baseline models, two traditional portfolio models that implement MPT (Markowitz's Modern Portfolio Theory) and Uniform Constant Rebalanced Portfolio (UCRP) strategy are employed. For the ablation study, simple Deep Deterministic Policy Gradient (DDPG), DDPG with Transformer (DDPG_TF), DDPG with 2D Relative- attentional Transformer (DDPG_RP_TF), and DDPG with Gated Transformer (DDPG_GL_TF) are tested under the same conditions. Both the cumulative return and the annualized Sharpe ratio are evaluated to verify the robustness of the models to risks as well as

their performances [52]. No further information was reported to better understand the nature and the possible performances of these models. For instance, we do not have any type of hyperparameter/model structure/references are written on the paper. This can represent a problem but it will be discussed later.

Model	Cumulative Return (%)	Annualized Sharpe Ratio
DPGRGT (Our Model)	**43.16**	**0.6418**
DDPG_GL_TF	11.93	0.2813
DDPG_RP_TF	-5.45	-0.1343
DDPG_TF	-41.71	-0.8191
DDPG	-42.91	-1.2194
UCRP	0.53	0.0125
MPT	-50.07	-1.5840

Figure 4.4: Results reported by Kim et al. [52]

Looking at these baselines the results obtained by Kim et al. looks really promising. The DPGRGT model overcomes abruptly all the baselines both as cumulative return or Sharpe ratio. If we compare this new result with the one of the standard transformer + DDPG the improvement is relevant. While the standard transformer has a negative final return the DPGRGT has positive cumulative returns of 43%. Also from the risk point of view, this new approach is the only way that widely overcomes the 0.3 thresholds.

4.2 The problems

In previous chapters, we have already raised some problems that affect this paper but, it was just the tip of the iceberg. Unluckily there are several issues and missing information that make the final result not replicable. Furthermore, some criticisms make the final outcome not trustable and difficult to interpret. Finally, we have to report that even they claim that the final portfolio is more realistic, due to the increased number of constraints, and have a good score both as cumulative return and Sortino ratio, no attempts on the real stocks market are done. It's a good exercise in style but, the lack of attempts in a real scenario resize the results that they obtain. We will come back to this problem later.

4.2.1 The input data

Let's proceed with the order and let's focus now on the input data. In previous chapters, we have analysed the complex distribution of the input data and the

problems that arose. Using only the original time series, the data results not normalized and even not stationary. Moreover, we highlighted relevant discrepancies between training and test data. All these problems adversely affect not only the performance but also the reliability of the ML tools. Several solutions were proposed to mitigate these obstacles like the introduction of the concept of returns. In Kim's paper, the data processing stage is poor and not clear. In the paper, the authors do not present an analysis on the behaviour or on the trend of the input data and they report only a line of preprocessing: "all the data is log differenced for the stationarity of the time series". This sentence is not only unclear but also conceptually wrong. To make time-series stationary, taking the log of the input prices is not enough. Maybe they want to say that they have used the log-returns (which are stationary), but it's not obvious. Now, let's come back to the problem of the 4d input data. As previous reported, Kim uses an input with an additional dimension to let the transformer learn relationships between the assets. This seems a brilliant idea but how the solution is implemented raises some doubts. During the modified self-attention layer, the input vector is reshaped to $(batch, H \cdot L, F)$. Where H is the number of assets, L is the window length considered and F is the dimension of the embedding space. With this solution, the transformer can learn relationships between different assets on different days. Even if conceptually right, practically this solution can not scale well with larger input data. The cost of computing the attention is quadratic with respect to the sequence of the input length. This means that if the size of $H \cdot L$ is large, the attention matrix becomes gigantic and intractable. So, if we want to learn long-term relationships, we have to decrease the number of input assets and vice versa. Using the hyperparameters reported in the paper, if we set $H = 10$ and $L = 50$ the size of the attention matrix is around 202k elements. This is not usual even in the Vision transformer paper [60], where an input image of $256x256$ is divided into mini-images of $16x16$ to reduce the computational effort. Let's introduced the last criticism related to the input data. First, we have to highlight that the trading days in a year are around 252. This depends on the fact that we exclude all the weekends and holidays. So, if the training period is around twenty years, we can easily compute the number of training samples as $20 \cdot 252 = 5040$. As you can imagine, this number of training samples is not enough to train an algorithm to solve a so complex task. Moreover, we are working with an RL agent that requires millions of samples to solve a simple task like Lunar Lander. This represents a major problem that it's difficult to face. They tried to mitigate this problem using a multi-simulation system but this increases only the experience of the RL and not the real number of samples.

4.2.2 Reward

Even if it's called p_{t-1}, this value represents the portfolio value at time t before having taken any action. In other words, we are at time t and before computing a_t, we look at the portfolio value. Then with equation (4.4), we compute the new s_t, namely we calculate how many stocks we can buy with a_t. The next step, with equation (4.6), is to compute the total transaction costs. This quantity b is then subtracted to the cash value of our portfolio s_t^0. As already said, if the transaction fees b are greater than the liquid cash, we can obtain $s_t^0 < 0$. This scenario is not ideal but from a practical point of view is not terrible. In a realistic scenario, the real agent can invest more money to avoid this problem or can sell a stock to avoid this situation. The real problem arises when they try to compute equation (4.10). p_t has highlighted in the previous chapters has a different meaning with respect to p_{t-1}. In fact, p_t is the value of the portfolio at time t after computing action a_t. p_t and p_{t-1} differs only for the second term. Using this confusing notation what it plays out is that $p_{t-1} \geq p_t$. This happens because to compute the portfolio value at time t we have spent the transaction fees. In this way the return r_t will always be a negative number and the RL algorithm will only focus on reducing the transaction costs.

Finally, let's consider the reward function chosen by Kim et al. namely the Sortino ratio. Taking their definition: "Sharpe ratio is defined as the average of historical returns from r_1 to r_t divided by the standard deviation of all the returns, whereas Sortino ratio is the expected return divided by the standard deviation of negative returns." Since the DDPG requires a reward for each action computed the Sortino ratio is calculated day after day. The problem that arises with this formulation is the following: suppose that you have a portfolio value of 100000 at time $t = 1$, then you compute some action and the portfolio goes to 110000 at time $t = 2$. Using equation (4.10) again, we have a return that is strictly greater than zero. Then, when we try to compute the Sortino ratio we discover that is impossible. This is because the standard deviation of negative returns does not exist and we can not divide the expected return for zero. Using this solution, the algorithm can not converge. The question now is: why they do not have this problem? Why did they not report that? The answer is very straightforward and it's based on what we have already said: the quantity r_t, in their formulation, can not be negative so, they can always compute the Sortino ratio. Obviously, this approach is wrong and can really affect the reliability and the result of the RL tool. What it's strange is that all these problems seem to not damage the final performances but, we will see in a moment why.

4.2.3 Markov decision process

Let's focus on the main problem that affects the RL settings. "A single observation set s_t s a three-dimensional tensor that consists of five features, Opening, High, Low, Closing prices (OHLC), and trading volumes of assets at time t"

$$s_t = (O_t, H_t, L_t, C_t, V_t) \tag{4.14}$$

This is how they have defined a state at time t. At first, this solution looks fine to face partial observability, but this approach has two dominant drawbacks. The first drawback is acceptable and starts from the assumption that since these are historical data, our actions can not have an impact on the time series. For this reason, this model can be used only by small-medium traders that, with their investments, have only a little or no impact on the market. This assumption is reasonable since we do not know deterministically how our actions can affect a stock price. The second drawback abruptly influences the final result of the model. Using only historical prices (OHLC) and trading volumes our actions can not affect the next state. In other words, starting from state s_t regardless of which action we choose, the next state will be with probability 1 $s_{t+1} \forall a \in A$ where A is the set of all possible actions.

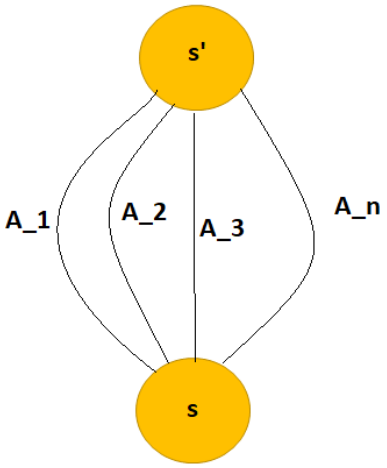


Figure 4.5

So, let's recap the Bellman equation describing the optimal action-value function $Q^*(s, a)$. It's given by:

$$Q^*(s_t, a_t) = E_{s_{t+1} \sim P}[r(s_t, a_t) + max_{a \in A}\, Q^*(s_{t+1}, a)] \tag{4.15}$$

In this case, we have not sampled s_{t+1} from a distribution P because, as we said before, the next step is deterministic. Moreover, since we take the max, with respect to the action set, $Q^*(s_{t+1}, a')$ is practically a constant. So, the only term that the algorithm can maximize is the reward function $r(s_t, a_t)$. This fact leads to a suboptimal solution. The algorithms will tend to prefer action that takes a large reward in a low term at the expense of larger rewards in the middle-long term. In general, the model should be able to take action that has a small reward in the short term but end up in states with a high long term reward. Practically, it's become a greedy approach. As we know, this type of solution is usually sub-optimal and can be really far away from the best one. Let's continue with another formulation issue that affects this MDP. In an RL setting is quite usual to have the training split into episodes. Usually, each episode starts with the same initial conditions and ends when the agent fails in achieving the final goal or when it overcomes the maximum number of actions for the episode. In the former case, the agent has to understand and remember the bad actions computed that lead to a failure. In the latter case, the highest reward possible is fixed and the RL agent has to learn how to achieve it at every episode. In their solution, Kim et al. decided to set the maximum length of a single episode at 50 days. As already said, this approach is allowed but, in this case, presents some drawbacks. In fact, with this formulation is impossible to establish when the RL problem is solved. For instance, the "Lunar lander" is considered solved when we achieve a score of 200 points when the episode length is set to 1000. So, for instance, we could decide that the problem is solved when the agent is able to earn 10k dollars every episode. But, suppose that in an episode all the stocks are losing economic value, how the agent can overcome the 10k dollars threshold? With this formulation is impossible to set any kind of baselines and the concept of episode solved loose of meaning. We have to highlight another significant issue related to this definition of an episode. To understand this other criticism, we have also to introduce how the test of the RL is developed. Usually, when we train an agent in an environment and we define the maximum number of steps, we have the same conditions in the test environment. Instead, Kim test the RL agent in a period of time that is much, much longer than 50 days, namely two years $\approx$ 500 days. When the agent is fully trained he knows that the first states s_0, s_1 and so on will tend to have a good value function $V(s_0), V(s_1)$ since there are a lot of opportunities to make money. Instead, when the agent approaches the last states of the episodes s_{48}, s_{49}, the corresponding value functions will have lower values than the start even if action a_{48} and a_{49} could be very good actions. Now suppose that during the testing period $s_{220} \approx s_{48}$, the agent will suppose to be near to an ending state and it will try to avoid it. But, maybe, s_{49} is not a bad state but, the algorithm will consider that in this way only because it was an ending state during the training period. This problem is more evident for terminal positions but, it's present in initial/middle states. What can cause further

instabilities even during the training period is the fact that the state of day 50 is very similar to the state of day 51. As we said the former will have a low-value function while the following day a very high one. Since we are not using a table to remember all the possible configuration of $Q(s, a)$ but a value approximator (namely a neural network), the latter will find major problems to understand if the state corresponding to days 50-51 is a good or a bad one. Finally, we have to highlight a weird strategy used during the training phase. In environments like Lunar Lander, once the episode is finished, the agent starts to explore from the beginning. This behaviour is repeated until the convergence of the method namely when the environment is solved. Even in the Kim implementation, we have several episodes that long 50 days. The problem is that when we arrived at the end of the training phase (namely at the end of the year 2018) the algorithm does not start again from day one. In other words, the RL agent is trained only once in the period 2000-2018. This solution will lead to a sub-optimal result due to a low exploration of the environment. For instance, let's consider investing X money on asset 1 on the day t. That particular action leads to a return of positive return Y. Now, having a positive value for Y is enough to establish that this particular action is good? We state no. Consider, this time, to invest X on asset 2 again on t. If this new action performs a return Z such that $Z > Y$ the previous action was not the best one. The only way, for the agent, to learn that the latter action is better than the former one is to try a lot of different actions on the day t. This is not possible on Kim's implementation where only one time we go through time t. Kim partially solved this problem using the multi-simulation approach but, using his parameters only 5 actions are selected at each time-stamp. For sure this helps but, it's not enough to achieve an optimal solution. This problem becomes more evident if we increase the number of assets taken into accounts.

4.2.4 Results

Finally, let's have a look again at their result. At a first look, they seem encouraging and incredibly overcomes any other baselines both as cumulative return and Sortino Ratio. The problem with all these baselines is the absence of any type of information. In fact, in the standard DDPG, we do not have any indication of the type of network used. Is it a normal multilayer perceptron? Is it an LSTM? Unluckily in the paper, the authors did not report that so, it's very difficult to understand the goodness of the model used. Moreover, for each baseline, we do not have any feedback about the hyperparameters used. It could be cool to have a link to other papers or some tables that gives more information about the type of networks and hyperparameters chosen. In this way, it's difficult to understand if the poor result obtained depends on the model itself or on a wrong choice of the parameters. Moreover, due to this lack of information, it is impossible to repeat the result obtained by these

baselines. Now, let's focus on the result achieved by the DPGRGT. Even if it's look incredibility high is hard to interpret due to the lack of standard deviations (STDs). Why this important information is missing? The results obtained in Figure 4.4 are obtained through "cherry-picking"? Or the STD is not reported because the method is so robust to noise and hyperparameters changes? Unluckily we do not have a clear answer to these questions and this fact affects badly the repeatability of this paper. The last point is related to a missing graph/result. What is really important in this task is to find the best portfolio to obtain a great result. This portfolio could change over time or could remain stable. In the former case, we understand that the agent prefers to pay higher transaction costs but try to exploit new market conditions. In the latter case, the agent instead has understood a strategy that performs well in the long period avoiding paying higher transaction fees. Both strategies are legitimate but have to be addressed. Kim et al., instead, reported neither the portfolio allocation nor how it changes over time. Practically, they have reported only half of the final result.

4.2.5 Missing information

Other problematic points are related to the poor attention dedicated to some details. In fact, in the last part of the actor and critic network (Figure 4.3) there are some dense layers in which we do not have any feedback on the number of neurons used and how the data is reshaped from a layer to another. After discussing with the authors, we understood that the output of the transformer is reshaped from (batch, H, L, F) to (batch, H * L * F). Using their hyperparameters we obtain an output of shape (32, 10 * 50 * 128) = (32, 64000). In ML is difficult to see a layer of this type of dimension and it's hard to understand the reasons for a so brutal flatten operation. Unluckily there is an additional problem. If the next layers had 4096 neurons, the total number of parameters would achieve $\approx$ 262 million. In one layer, they would have the equivalent of four Resnet152. This implies that the majority of parameters and the discriminative power is located in only one layer. This solution is hard to understand and becomes extremely difficult to train a so deep model. After this calculation, a natural question that arises is: how they train a so large model on the basic version of Google Colab? Colab has not so much RAM available and training a deep transformer like this is almost improbable. Unluckily there isn't an implementation available so, it is practically impossible to replicate this paper.

Figure 4.6: Two other portfolio strategies as additional baselines.

Even in the case, this paper was perfectly replicable, we will introduce two other baselines that highlight the poor results obtained by Kim. Let's consider Figure 4.6. In this graph, we have reported two different portfolios that follow opposite strategies. The orange curve represents an equally weighted portfolio, while the blue line represents a portfolio where we have invested only in the "AAPL" stock. As you can see, the former portfolio constituted of the 10 initial stocks performs not so well but, it's practically better than all the baselines chosen by Kim. Even a very naive strategy like this one is more competitive than most of the algorithms in Kim's paper. This highlight the fact that they select not performant baseline to compare with their model. This fact is even more evident if we consider the second portfolio. Using the only "AAPL" approach, the cumulative result obtained is more than twenty per cent higher than the one achieved through the DPGRGT. This further analysis clearly shows that the performance of Kim's model is not at the level they claim.

With all the issues replicate this paper was an enormous challenge. There are also other minor problems related to how attention and some formulas are computed, but they are only a drop in the sea. Having said that we tried to implement the DPGRGT model but, after few attempts, it was clear that be able to obtain a similar result was almost impossible. Unluckily the unwillingness of the authors to share the code and further information makes this paper not replicable. But after all the problems encounter maybe, the result they obtained is not meaningful but could represent a good starting point to start our work.

4.3 New Proposal

4.3.1 Introduction

Based on the idea of Kim et al. and Parisotto et al., we present our version which maintains similar settings adding some variations to fix the aforementioned problems. The main points of interest regarding how to preprocess the input data, a new formulation of the MDP, a new RL algorithm and the usage of TRLX-I over

GTRXL. As we will see these modifications will lead our model to obtain significant results that outperform the Kim et al. method. Moreover, some ablation studies will be reported to underline the improvements of this method with respect to other architecture like standards transformer, LSTM and RL agents. The ablations studies will confirm our choices and will help us to understand why this method is able to achieve some good results.

4.3.2 The input data

"Portfolio Selection" is a methodology that is studied in several papers [1] [61] [62] for a long time. Choosing the right assets to trade on is a complex problem like select the right strategy to follow. If we select firms that in the future will default, even for the best algorithm will be hard to make a profit. On the other hand, if the selected firms have a value that always increases in time, the final result will be meaningless. This thesis does not focus on the "Portfolio Selection" part so we will decide the training and test assets in a way to guarantee consistent results. We leave this selection stage to further analysis and papers and this, for sure, will help this algorithm to additional improvements. Now, before deciding the assets that will be inside our portfolio, we have to highlight a relevant fact. When we train these kinds of models with multiple firms is difficult to understand if they are learning something or instead they are deciding at random. Especially if we select assets that grow a lot in time, even random actions can easily lead to a huge profit. To let the reader understand this point, we report a curious result in Figure 4.7.

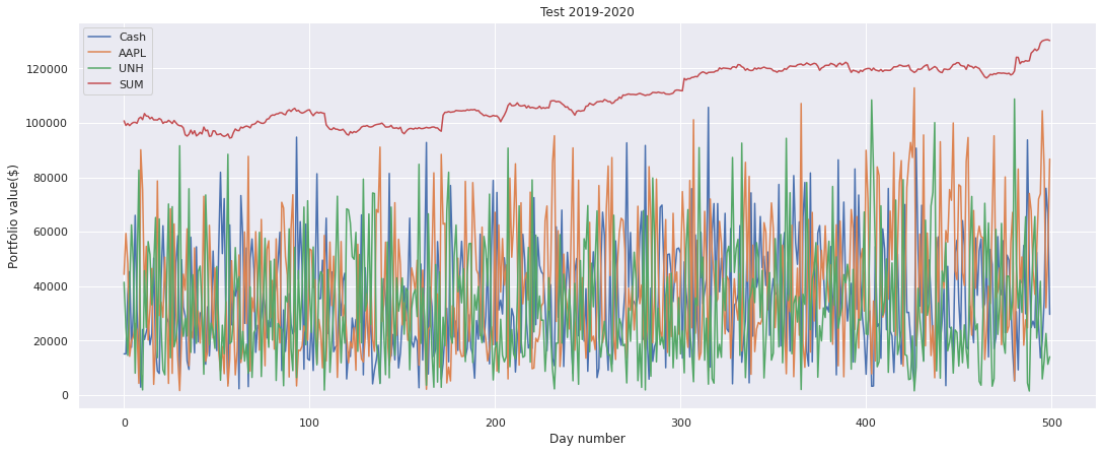


Figure 4.7: Random agent working in the period 2019-2020 on AAPL, UNH and cash.

In this chart, we show that using only two assets (AAPL and UHN) plus cash and a random strategy, we are able to obtain a cumulative return of around 30%.

So, increasing the number of assets not only make the problems more complex for the RL agent but, also understand its behaviour becomes incredibly hard. Only if the agent starts to have losses we can assume that it is not performing well. The question now is how we can distinguish if Figure 4.7 is the result of a trained agent or a random one? Unluckily we have not found a precise answer to this problem. So, how we can be sure of our final results if we do not have if a graph is the output of a clever strategy or a trivial one? The approach followed to validate our model is the following: we train and our model using only one asset per time. We try different firms to be sure that the results are competitive. We used the model and hyperparameters of the previous point to train the model in a multi-asset setting. Even with one asset per time is possible to earn money with a random strategy but, it's more simple to understand if the model is learning or not. In fact, we can focus our attention on the points where the asset is losing value. For instance, if the RL agent is able to spot that points and save money this means that the model is learning meaningful features. Another relevant problem related to the Kim et al. version was the low number of sample data. Kim, for the training set, used only data from 2000 to 2018 and test on the period 2019-2020. Two years for the testing period seems reasonable but, we can not say the same for the training one. Luckily assets like Apple, 3M are alive starting from the 1980s. This means that to extend the training period we can select firms that are older than the 2000s. This signifies that we are discarding earlier companies but, only this kind of choice can be done to increase the dimension of the training set. For our setting, we have decided to select assets that in the period 2019-2020 has both earn or lose value. This is done to understand if the agent can pick the right firms in which to invest. At this point, a new issue emerges. The assets that are survived from the 1980s to the 2000s, in general, has increased dramatically their values. We can have some short decreasing or stable trends, but they are limited. So, it's become really difficult to train a model on an asset that increases a lot in the training and in the test drops significantly. In fact, if the training data are biased this type of bias will propagate also in the testing phase. This is a significant problem that is difficult to remove due to the trading dynamics. What could happen is that the RL agent is too optimistic in the testing forecasts. We have to check if this type of behaviour will occur. Finally, as already introduced in the first stages of this thesis, we will not use the usual time series but, we will adopt the log returns. This is done to guarantee normalized and stationary input data.

4.3.3 Markov decision process

In the previous chapter, we have analysed all the issues related to the implementation of Kim et al. and how this formulation could lead to a suboptimal result. Their solution focuses its attention on seeking the best action at each timestamp t without

looking at their actual portfolio allocation. This approach can only adversely affect the final performances of the RL agent. Knowing the actual portfolio is crucial to help the agent to minimize the transaction costs and optimize the final return. To clear this point, briefly, we will report an example. Suppose your agent realizes that Apple's value will increase significantly its value in a month. In the Kim implementation, the agent, regardless of its portfolio, decides to invest everything in Apple's stocks. This action could be cheap if part of the cash is already invested in "AAPL" or incredibly expansive if the agent does not have any stock of that asset. Kim's agent is not able to understand a priori if the following action will imply massive transaction costs or not. This happens because the state seen by the agent does not contain any information on the actual portfolio allocation. To be able to select the correct action a_t, the agent must know the portfolio distribution s_{t-1}. This approach implies that depending on the action selected at every time t, the RL agent can end up in a different state like s', s'' and so on. This new solution is more complete but increases exponentially the RL problem complexity. In Kim's implementation, the starting and ending states are fixed, while several actions can be chosen. In this new implementation, starting from a defined state, different actions can be select and each action leads to a different new state. Finally, we are coming back to a more normal setting for an RL problem, at least from the actions/states perspective. Another significant issue that affects Kim's implementation concerns the definition and the length of an episode. Their idea, as highlighted in the previous chapters, could create problems in the estimation of the action-value function $Q(s, a)$. To overcome this problem, we have completely changed their approach based on a real case scenario. When we train an RL algorithm like that, we want to elaborate a strategy for the middle-long term period. So, the testing period is not pre-defined and can change depending on the need. For this reason, all the training period corresponds to a single episode. Each episode is repeated multiple-time until the RL agent solves the problem. An episode is considered solved depending on the needs and the objectives of the users. Obviously, to obtain better results a larger training period is needed.

Reward function

The reward function, conceptually, will be very similar to the one introduced by Kim et al. but, it will have a major impact on the final result. The main idea behind this new function is to try to reduce the enormous complexity of this problem at the expense of the final result. All the modifications introduced in the previous chapter makes this task too difficult to solve using the little amount of data available. So, even in this solution, only long actions are allowed. We prefer long over short positions due to the higher possibility of making a profit. For this reason, we maintain the formula (4.22) to define an action. Since it is impossible

to know exactly when we will make a transaction during a day, the value of each asset is computed by averaging the OHLC. For each asset i at time t:

$$value_t^i = \frac{O_t^i + H_t^i + L_t^i + C_t^i}{4} \tag{4.16}$$

Now we compute the portfolio value as:

$$p_{value_t} = value_t * s_{t-1} \tag{4.17}$$

To compute the new portfolio s_t, giving the action a_t:

$$s_t = (p_{value_t} * a_t)//value_t \tag{4.18}$$

Then we apply the transaction fees as:

$$b_t = \sum_{i=1}^{m} value_t^i \left| s_t^i - s_{t-1}^i \right| \cdot 0.2 \tag{4.19}$$

At this point we can compute how much cash we will have at time t combining all these concepts:

$$s_t^0 = p_{t-1} - \sum_{i=1}^{m} c_t^i s_t^i - b_t \tag{4.20}$$

Then we can calculate the portfolio value at the end of day t as:

$$p_{end} = C_t * s_t \tag{4.21}$$

Finally, to understand if the action computed has increased our portfolio value with respect to the opening values we calculate:

$$p_{start} = O_t * s_{t-1} \tag{4.22}$$

The reward function is simply:

$$r = \frac{p_{end} - p_{start}}{p_{start}} \tag{4.23}$$

This formulation is much more intuitive with respect to the previous one and the approximation done are introduced without losing generality. With this reward function, the risk is not taken into account directly. In this way, the algorithm could find risker solutions but the task becomes easier for an RL agent. Finding a correct and proper formulation to take into account the risk is left to future work.

4.3.4 Architecture

In Parisotto et al. [35] there are three variants of the transformer: the TrXL, the TrXL-I and the GTrXL. Based on the results obtained, the first model (TrXL) is unable to solve the human normalized task with a score of 5.0. The TrXL-I using the "Identity Map Reordering" is able to solve the task with a mean reward of 107.0. Finally, the GTrXL, which introduces a GRU level to replace the sum operation, achieves an average reward of 117.6. What is clear is that reordering the skip connection in the TrXL-I is the real trick that stabilizes the model and makes it converge. It is not clear whether the GRU gate could always be helpful. In fact, in their studies, they tested several recurrent architectures that negatively impact agent performance. For this reason, it is not clear whether the GRU level could be useful for other tasks such as that of this thesis. In addition, the recurrent mechanism is introduced to alleviate partial observability of the model and remembers stuff that may be useful in future steps. In our case, this mechanism is not necessary. In fact, in our input data, we have already passed a few days of history for each asset, this makes the GRU layer redundant. For these reasons the TrXL-I could be the most suitable layer for this new task.

Model	Mean Human Norm.	Mean Human Norm., 100-capped
LSTM	99.3 ± 1.0	84.0 ± 0.4
TrXL	5.0 ± 0.2	5.0 ± 0.2
TrXL-I	107.0 ± 1.2	87.4 ± 0.3
MERLIN@100B	115.2	89.4
GTrXL (GRU)	117.6 ± 0.3	89.1 ± 0.2
GTrXL (Input)	51.2 ± 13.2	47.6 ± 12.1
GTrXL (Output)	112.8 ± 0.8	87.8 ± 0.3
GTrXL (Highway)	90.9 ± 12.9	75.2 ± 10.4
GTrXL (SigTanh)	101.0 ± 1.3	83.9 ± 0.7

Figure 4.8: Transformer's performance comparison [35].

4.3.5 Reinforcement Learning Agent

The RL algorithm chosen by Kim was the DDPG. This method works well in different types of tasks, as we have already seen in the previous chapters. As reported, we were not able to reproduce their results so, it was complex to evaluate the goodness of this architecture choice. In our previous experiments reported in Chapter 2.8.3, we have seen that the SAC algorithm often outperforms the result obtained by the DDPG one. So, to improves the Kim et al. [52] performances, we first start with DDPG and then we switch to the SAC. Starting from the first experiments, what becomes immediately clear was that the results obtained by DDPG were extremely poor. The new method was not able to learn any meaningful

pattern. After few epochs the algorithm starts to converge to a suboptimal solution like all in one asset Figure 4.9.

Figure 4.9: The new proposal with the DDPG model test on three assets.

Initially, we suppose to have some problems with the input data or with the transformer but everything seems to work fine. So, thanks to the literature, we focus on understanding why DDPG has this unusual behaviour. In the papers [63] [64] [65], the authors underline some shortcomings of DDPG and show why the ddpg algorithm fails to achieve convergence in some situations:

- The DDPG is designed for settings with continuous and often high-dimensional action spaces and the problem becomes very sharp as the number of agents increases.
- The second problem comes from the inability of DDPG to handle variability on the set of agents. Since we use five different agents as the Kim paper, this could adversely affect the performance of this model.
- The settings in real-life platforms however are quite dynamic, with agents arriving and leaving or their costs varying over time, and for an allocation algorithm to be applicable, it should be able to handle such variability.
- DDPG needs a lot of steps to converge. As shown in previous experiments this algorithm can require an incredible amount of episodes to finally converge to the desired solution. Moreover, in this setting, the amount of data available is extremely limited.
- Finally most of the algorithms that use the replay memory do not perform well in a changing environment because the model can not generalize properly

from past experiences and these experiences are used inefficiently. In addition, these problems are raised where the increasing number of agents expands the state-action space exponentially.

After some experiments and this in-deep analysis, we prefer to use all the time resources on the SAC variant. As we have seen and we will see this algorithm looks more promising to obtain the aspirated result.

4.3.6 Hyperparameters table

Hyperparameter	Value
Batch-size	32
Discount factor (γ)	0.99
Number of layers	2
Layers hidden size	64
Number of multi-heads	8
Number of traders	5
Window size	50
Number of training days	7055
Number of test days	502
Fees for transaction	0.2%
Replay buffer size	$4e+6$
Learning rate actor	$1e-5$
Learning rate critic	$1e-4$
Learning rate temperature	$3e-3$
Polyak update value (τ)	0.95

Chapter 5

Results and Conclusions

5.1 Introduction

As previously reported, it is very complex to understand if a model can learn the complex patterns and relationships among different assets. How can we know if our model behaves following a good policy or a random one? As already seen in Figure 4.7 even tossing a coin, we can obtain a great result. So, our strategy was to start with a single asset (and cash) per time. Using a simpler environment like this one can be helpful to understand if our model can learn meaningful features about the asset behaviour. As we will see when the number of companies starts to increase understanding agent's actions becomes complex. According to the previous chapter, finding baselines for this task is not easy. We have not found any paper that reproduces the following experiments using RL agents. For this reason, we have decided to introduce two different baselines that can help the reader to understand better the result that we have obtained. The methods used are:

1. An equally weighted portfolio. So, if our portfolio is composed of 'Cash' and 'Apple' stocks, 50% of the money are invested in 'Cash' and the other 50% in 'Apple'. This strategy even if very naive, does not require a lot of transactions and this will translate into fewer fees to pay.

2. A competitive random portfolio. With competitive, we mean that we run a random simulation until the final portfolio value overcomes the 10% of revenues. If after twenty runs no random strategies can overcome this threshold, no result is reported.

These baselines are not so competitive but, if we can overcome them, we have finally designed a model that can produce a valid result. For our initial experiments, we focus only on three different assets that work in very different fields like: 'AAPL' (Apple Inc.), 'UNH' (UnitedHealth Group), '3M' (3M Company). For the sake of

simplicity, the initial amount of invested money is set to 100k $. In the test period, these assets have very different behaviours. The asset prices can change a lot with fluctuation larger than 20%.

5.2 Results on AAPL

After a long hyperparameters tuning phase, we have trained our model based on the values that are reported in Table 4.3.6. The first environment takes into consideration only the Apple Inc. stock for the reasons explained in the introduction. We have done several experiments (with different seeds) and we report the mean portfolio value and the relative standard deviation in Table 5.1 computed from 5 different runs. In Figure 5.1, 5.2 and 5.3 (where the Total Portfolio value at time t is the sum of the values of all the assets at time t.) we have reported the strategies taken by the agent at the end of 3 different episodes. As you can see the agent improves his behaviour episode after episode. In Figure 5.1 the model struggles to make a profit and make a lot of useless transactions. Despite this fact, the model is able to waste money and ends with a portfolio value similar to the original one. In episodes 2 and 3 the model learns how to avoid expansive transactions and learn to spot interesting phenomenon. After the third episode, the agent is not able anymore to learn interesting features. If we extend the training period to 10 or more episodes the model tends to convert to a single solution: investing all the money in Apple Inc. stock at time $t = 0$ and then no other transactions are performed. For sure, this solution avoids expensive transaction costs but, the risk of taking this solution is higher. In fact, if Apple Inc. has a default, we risk losing a lot of money in a very few time. The result overcomes the two baselines that we have selected but to be sure that we are not in a lucky case, we propose another attempt, with a similar environment.

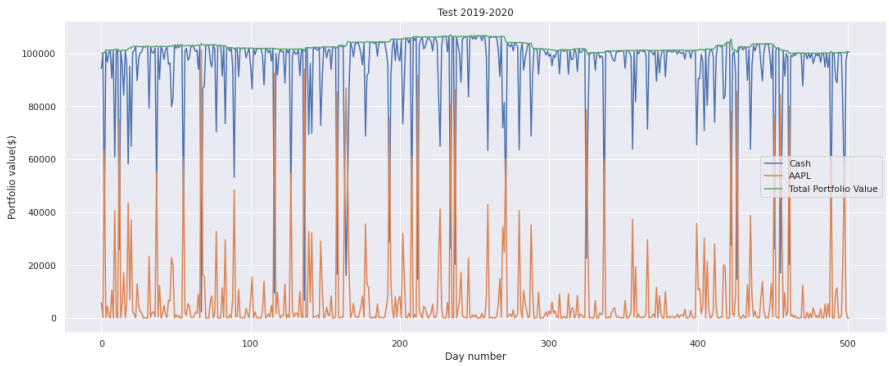


Figure 5.1: Episode 1 on AAPL and cash. Training period 1990-2018. Test period 2019-2020.

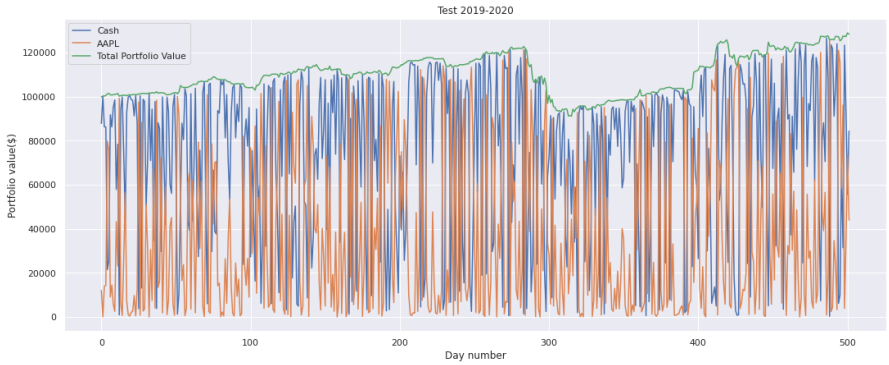


Figure 5.2: Episode 2 on AAPL and cash. Training period 1990-2018. Test period 2019-2020.

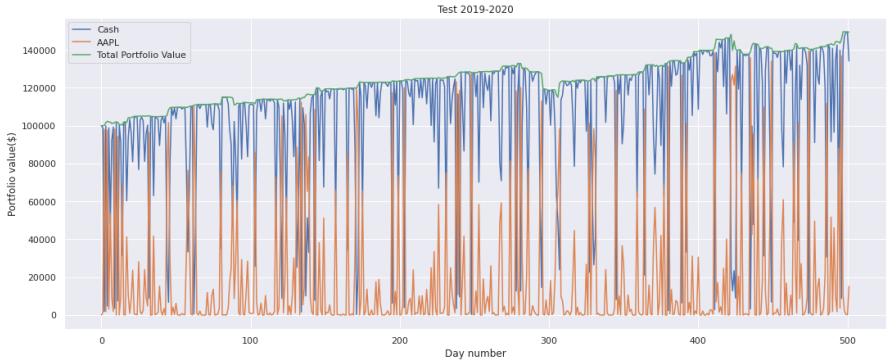


Figure 5.3: Episode 3 on AAPL and cash. Training period 1990-2018. Test period 2019-2020.

Table 5.1: Results comparison

Method	*Final portfolio value*
Our method	150327 ± 2441\$
Equally weighted portfolio	121612 ± 0\$
Random	110289 ± 0\$

5.3 Results on 3M

In this new setting, we have simply replaced the 'AAPL' stock with '3M'. The behaviour highlighted in the previous results repeats itself even with this new stock. In the first episode, the model makes only a few transactions and the final portfolio value is practically the same as the beginning. In the following episodes, the agent

improves its understanding of the stock pattern and is able to improve the final profit. As you can see from the results in Table 5.2 the result is not competitive as the previous one (table 5.1). The model outperforms our baselines but the improvement is not obvious as in the former result. This fact could suggest that the model could struggle to obtain returns even in a simple environment like this one. These difficulties can depend both on the low amount of data available or on the fact the states used by the RL agent are not Markovian. Basing our decisions only on the asset prices in certain situations could not be enough.

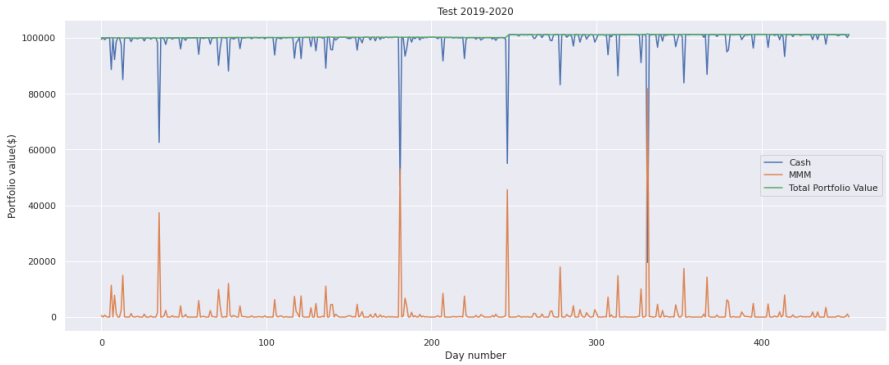


Figure 5.4: Episode 1 on 3M and cash. Training period 1990-2018. Test period 2019-2020.

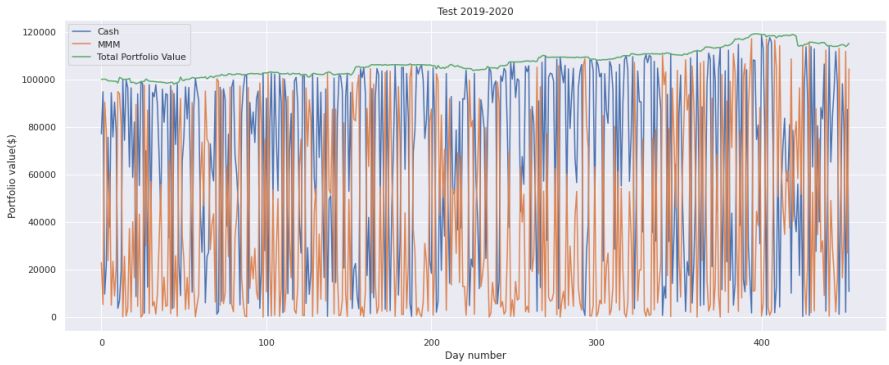


Figure 5.5: Episode 2 on 3M and cash. Training period 1990-2018. Test period 2019-2020.

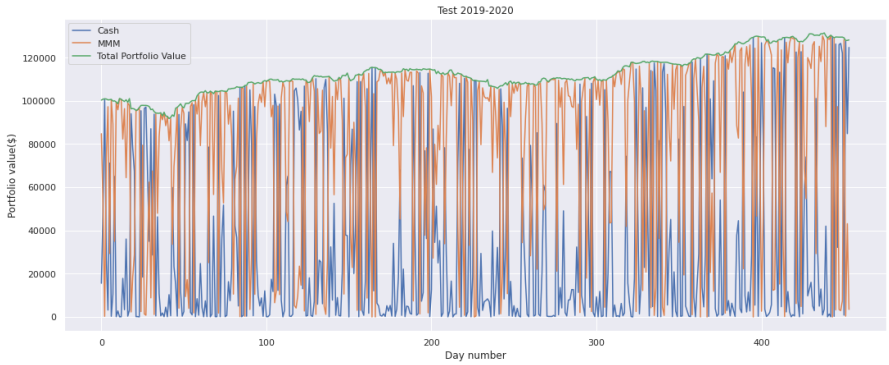


Figure 5.6: Episode 3 on 3M and cash. Training period 1990-2018. Test period 2019-2020.

Table 5.2: Results comparison

Method	*Final portfolio value*
Our method	125407 ± 4694\$
Equally weighted portfolio	119234 ± 0\$
Random	111256 ± 0\$

5.4 Results on AAPL and UNH

The following step was to try to increase the number of assets taken into account by the model and look if the model was able to learn the inner relations between the different assets. We start using 'CASH', 'AAPL', and 'UNH'. As you can see in figure 5.7 and 5.8 the model both in the worst and in the best case is able to guarantee a relevant profit. As is clear in table 5.3 even if the final mean profit is quite high the standard deviation is increased significantly from the previous results. As previously said when the number of assets starts to increase is very difficult to understand the inner behaviour of the agent. When the problem starts to become so complex the decisions taken by the agents during the training period start to weigh much more. Maybe for this task, a higher number of episodes could help the model to converge better to a unique solution. We have avoided increasing the number of episodes to make the results obtained more comparable to the previous ones.

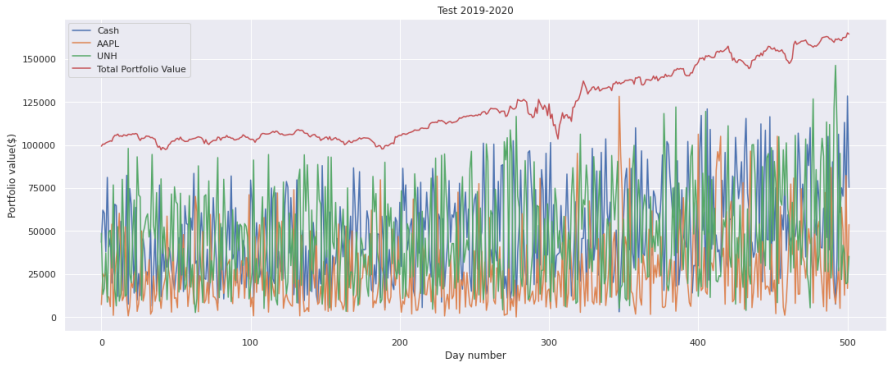


Figure 5.7: Best result on AAPL, UNH and cash. Training period 1990-2018. Test period 2019-2020.

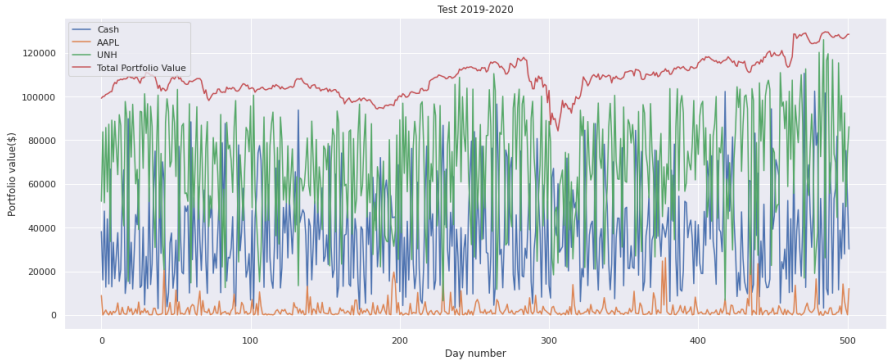


Figure 5.8: Worst result on AAPL, UNH and cash. Training period 1990-2018. Test period 2019-2020.

Table 5.3: Results comparison

Method	*Final portfolio value*
Our method	145816 ± 12098\$
Equally weighted portfolio	131407 ± 0\$
Random	110912 ± 0\$

5.5 Results on AAPL and 3M

As already done in the previous step we have trained other models using different stocks to check if the results obtained in the previous attempt be reliable. In this case, we have chosen 'CASH', 'AAPL' and '3M'. The result, as you can see in Table 5.5, is similar to the former one. The agent can overcome the baselines but, the

final standard deviation is quite relevant. We have also to highlight that finding a strategy for this portfolio was also tougher than in the previous cases. After 20 random simulations, no strategy leads to a profit greater than 10%.

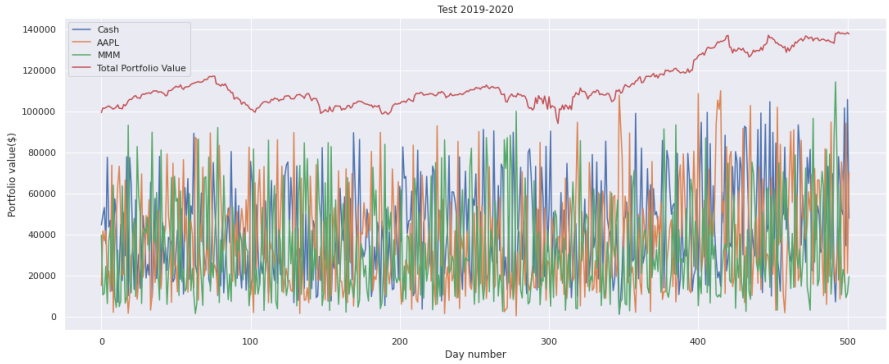


Figure 5.9: Best result on AAPL, 3M and cash. Training period 1990-2018. Test period 2019-2020.

Table 5.4: Results comparison

Method	*Final portfolio value*
Our method	134227 ± 5718\$
Equally weighted portfolio	126127 ± 0\$
Random	-

5.6 Results on more than three assets

The last step was to try to train our agent to deal with a more extended set of assets as done in [52], [66]. In our experiments, we have tried both with five, ten and twenty assets. In all these experiments after only a few steps of the first episode, the network diverges and starts to invest only in a single asset. This fact suggests that the problem becomes too difficult to handle for the network. The latter to avoid massive transaction fees prefer to invest in only one asset. To fix this problem, we have tried to change the α (temperature) parameter of the SAC algorithm to prefer exploration over exploitation. Even with extreme values of α, the final result was the same. The last attempt was to try to make our Transformer deeper increasing the number of layers as done in [35]. The model seems to behave better but, after four/five episodes, the results were not so competitive. Increasing the complexity of our network could be a solution to this problem but, the long time to try our model on Google Colab GPUs made additional attempts impossible. This attempt is left to further analysis and studies.

5.7 Results review

Table 5.5: Results comparison

Method	*Results on AAPL*	*Results on MMM*	*Results on AAPL-UNH*	*Results on AAPL-3M*
Our method	150327 ± 2441\$	125407 ± 4694\$	145816 ± 12098\$	134227 ± 5718\$
Equally weighted portfolio	121612 ± 0\$	119243 ± 0\$	131407 ± 0\$	126127 ± 0\$
Random	110289 ± 0\$	111256 ± 0\$	110912 ± 0\$	-

5.8 Conclusions

Finding the algorithm that can always earn money is the Holy Grail for every broker, banker and trader. As we stated in previous chapters, it is an extremely complex problem from different points of view. The low amount of data represents a massive difficulty for almost all machine learning algorithms. Moreover, the market behaviour changes periodically and handling this phenomenon is quite difficult for every type of ML architecture. To be able to understand how an asset could behave in the future, a lot of additional information must be taken into account. For instance, the company's financial statements, news, insights and so on. To handle, this type of data and information several types of architectures and computational power is required. This thesis aimed to study the state of the art in this field and propose a variant that could increase the current baseline. As we have seen, the results obtained in previous works like [52], [66] are not so competitive and could not be reproduced. For this reason, we have tried to learn from this previous experience and we have developed a new model based on Transformer and Reinforcement learning. What is clear from the results at the start of this Chapter 5 our model can be competitive in very naive scenarios. The model can produce remarkable outcomes if it works with a reduced number of assets. Instead, several problems arise when we start to deal with five or more stocks. This highlights the fact that the network that we have designed is not so stable and it is very complicated that could be used in real case scenarios. In fact, one of the major issues to use RL and ML is the difficulty to produce explainable results. When several stocks are taken into account is very complicated to understand if the network is producing a valid result or if the final result depends only on lucky decisions. From a performance point of view, the method developed seems competitive but to use in a real scenario further analysis must be computed to understand the validity of the result produced. This problem is common in the machine learning community and several techniques could be used.

Bibliography

[1] Harry Markowitz. «PORTFOLIO SELECTION*». In: *The Journal of Finance* 7.1 (1952), pp. 77–91. DOI: https://doi.org/10.1111/j.1540-6261.1952.tb01525.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-6261.1952.tb01525.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1952.tb01525.x (cit. on pp. 1, 93).

[2] Angelos Filos. *Reinforcement Learning for Portfolio Management*. 2018. URL: https://arxiv.org/pdf/1909.09571.pdf (cit. on p. 1).

[3] Yiyong Feng and Daniel Palomar. «A Signal Processing Perspective of Financial Engineering». In: *Foundations and Trends® in Signal Processing* 9 (Jan. 2016), pp. 1–231. DOI: 10.1561/2000000072 (cit. on p. 4).

[4] Angelos Filos. *Reinforcement Learning for Portfolio Management*. 2019. arXiv: 1909.09571 [q-fin.PM] (cit. on pp. 4, 6).

[5] James Chen and Thomas Brock from Investopedia.com. *Risk*. 2020. URL: https://www.investopedia.com/terms/r/risk.asp (cit. on p. 8).

[6] Jason Fernando and Margaret James from Investopedia.com. *Sharpe Ratio*. 2021. URL: https://www.investopedia.com/terms/s/sharperatio.asp (cit. on p. 9).

[7] Reinforcement Learning Course by David Silver. *LSTM*. 2015. URL: https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&ab_channel=DeepMind (cit. on pp. 10, 14).

[8] David Silver. *David Silver website: lessons about reinforcement learning*. URL: https://www.davidsilver.uk/teaching/ (cit. on pp. 10, 12, 16, 17, 22, 24, 26–28, 31, 34, 35).

[9] RAIL. *CS 285*. 2021. URL: https://www.youtube.com/watch?v=JHrlF10v2Og&list=PL_iWQOsE6TfXxKgI1GgyV1B_Xa0DxE5eH&ab_channel=RAIL (cit. on p. 10).

[10] Abhijit Gosavi. *Simulation Based Optimization Parametric Optimization Techniques and Reinforcement Learning*. 2015. DOI: 10.1007/978-1-4899-7491-4 (cit. on pp. 10, 13, 53, 57).

[11] Haiyan Chen and Fuji Zhang. «The expected hitting times for finite Markov chains». In: *Linear Algebra and its Applications* 428.11 (2008), pp. 2730–2749. ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2008.01.003. URL: https://www.sciencedirect.com/science/article/pii/S0024379508000244 (cit. on p. 12).

[12] spinningup-openai. *Part 2: Kinds of RL Algorithms*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (cit. on p. 39).

[13] Xin Xu, Dewen Hu, and Xicheng Lu. «Kernel-Based Least Squares Policy Iteration for Reinforcement Learning». In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 18 (Aug. 2007), pp. 973–92. DOI: 10.1109/TNN.2007.899161 (cit. on p. 42).

[14] Ankit Choudhary. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*. 2019. URL: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ (cit. on p. 43).

[15] Menghao Wu, Yanbin Gao, Alexander Jung, Qiang Zhang, and Shitong Du. «The Actor-Dueling-Critic Method for Reinforcement Learning». In: *Sensors (Basel, Switzerland)* 19.7 (Mar. 2019). ISSN: 1424-8220. DOI: 10.3390/s19071547. URL: https://europepmc.org/articles/PMC6479875 (cit. on p. 44).

[16] Mathworks. *Mathworks*. URL: https://it.mathworks.com/help/reinforcement-learning/ug/sac-agents.html (cit. on p. 44).

[17] spinningup-openai. *Soft Actor-Critic*. URL: https://spinningup.openai.com/en/latest/algorithms/sac.html (cit. on p. 46).

[18] spinningup-openai. *Deep Deterministic Policy Gradient*. URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html (cit. on p. 48).

[19] Antonin Raffin. *RL Baselines3 Zoo*. https://github.com/DLR-RM/rl-baselines3-zoo. 2020 (cit. on pp. 49–52).

[20] Antonin Raffin. *RL Baselines Zoo*. https://github.com/araffin/rl-baselines-zoo. 2018 (cit. on pp. 49–52).

[21] gym-openai. *LunarLander-v2*. URL: https://gym.openai.com/envs/LunarLander-v2/ (cit. on pp. 49, 50).

[22] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2021 (cit. on pp. 53, 58).

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: (2017). arXiv: 1706.03762 [cs.CL] (cit. on pp. 53, 57, 60, 61).

[24] Stanford University. *Stanford CS224N: NLP with Deep Learning – Transformers and Self-Attention*. 2019. URL: https://www.youtube.com/watch?v=5vcj8kSwBCY&t=870s&ab_channel=stanfordonline (cit. on pp. 53, 59).

[25] NERSC. *Symmetry and Equivariance in Neural Networks - Tess Smidt*. 2020. URL: https://www.youtube.com/watch?v=8s0Ka6Y_kIM&ab_channel=NERSC (cit. on pp. 53, 73).

[26] London Machine Learning Meetup. *Daniel Worrall: Understanding and Generalising the Convolution*. 2018. URL: https://www.youtube.com/watch?v=TlzRyHbWeP0&t=1411s&ab_channel=LondonMachineLearningMeetup (cit. on p. 53).

[27] Dr. Smitha Rao Saahil Afaq. *Significance Of Epochs On Training A Neural Network*. June 2020. URL: https://www.ijstr.org/final-print/jun2020/Significance-Of-Epochs-On-Training-A-Neural-Network.pdf (cit. on p. 56).

[28] Jason Brownlee from machinelearningmastery.com. *Overfitting and Underfitting With Machine Learning Algorithms*. 2019. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (cit. on p. 56).

[29] Xavier Glorot and Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html (cit. on p. 56).

[30] Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. *Normalization Techniques in Training DNNs: Methodology, Analysis and Application*. 2020. arXiv: 2009.12836 [cs.LG] (cit. on p. 57).

[31] Alex Sherstinsky. «Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network». In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306. URL: http://dx.doi.org/10.1016/j.physd.2019.132306 (cit. on p. 57).

[32] Anirudha Ghosh, A. Sufian, Farhana Sultana, Amlan Chakrabarti, and Debashis De. «Fundamental Concepts of Convolutional Neural Network». In: Jan. 2020. ISBN: 978-3-030-32643-2. DOI: 10.1007/978-3-030-32644-9_36 (cit. on p. 57).

[33] Amirhossein Kazemnejad. *Transformer Architecture: The Positional Encoding*. 2019. URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ (cit. on p. 63).

[34] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. «Self-Attention with Relative Position Representations». In: (2018). arXiv: 1803.02155 [cs.CL] (cit. on p. 65).

[35] Emilio Parisotto et al. «Stabilizing Transformers for Reinforcement Learning». In: (2019). arXiv: 1910.06764 [cs.LG] (cit. on pp. 67, 97, 106).

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV] (cit. on p. 68).

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. *Language models are unsupervised multitask learners*. 2019 (cit. on p. 68).

[38] Alexei Baevski and Michael Auli. Adaptive. *Input representations for neural language modeling*. 2019 (cit. on p. 68).

[39] Yiming Yang Zihang Dai Zhilin Yang. *Transformer-XL: Attentive language models beyond a fixed-length context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019. URL: https://www.aclweb.org/anthology/P19-1285 (cit. on p. 68).

[40] Samuel G. Finlayson. *Induction, Inductive Biases, and Infusing Knowledge into Learned Representations*. 2020. URL: https://sgfin.github.io/2020/06/22/Induction-Intro/ (cit. on p. 69).

[41] Hossein Gholamalinezhad and Hossein Khosravi. *Pooling Methods in Deep Neural Networks, a Review*. 2020. arXiv: 2009.07485 [cs.CV] (cit. on p. 71).

[42] Aurélien Géron. *Hands-on Machine Learning with scikit-learn, Keras and Tensorflow*. 2019 (cit. on p. 72).

[43] Daniel E. Worrall, Stephan J. Garbin, Daniyar Turmukhambetov, and Gabriel J. Brostow. «Harmonic Networks: Deep Translation and Rotation Equivariance». In: *CoRR* abs/1612.04642 (2016). arXiv: 1612.04642. URL: http://arxiv.org/abs/1612.04642 (cit. on pp. 73, 74).

[44] Tess Smidt. *Symmetry and Equivariance in Neural Networks*. 2020. URL: https://docs.google.com/presentation/d/1Acz3YxUI-pH80n4UOeHWktnnpjslsTXkilmP5ZGBNo4/edit#slide=id.g938c0ac37a_0_5497 (cit. on p. 74).

[45] Yann LeCun and Corinna Cortes. «MNIST handwritten digit database». In: (2010). URL: http://yann.lecun.com/exdb/mnist/ (cit. on p. 74).

[46] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «Imagenet: A large-scale hierarchical image database». In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 74).

[47] Jason Brownlee. *No Free Lunch Theorem for Machine Learning*. 2021. URL: https://machinelearningmastery.com/no-free-lunch-theorem-for-machine-learning/ (cit. on p. 74).

[48] Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V. Le. *Attention Augmented Convolutional Networks*. Oct. 2019 (cit. on p. 76).

[49] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 76).

[50] Pytorch. *LSTM*. URL: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html (cit. on p. 77).

[51] Ke Xu, Yifan Zhang, Deheng Ye, Peilin Zhao, and Mingkui Tan. *Relation-Aware Transformer for Portfolio Policy Learning*. 2020. URL: https://doi.org/10.24963/ijcai.2020/641 (cit. on p. 78).

[52] Tae Wan Kim and Matloob Khushi. *Portfolio Optimization with 2D Relative-Attentional Gated Transformer*. 2020. arXiv: 2101.03138 [q-fin.PM] (cit. on pp. 78–83, 85, 97, 106, 107).

[53] Bin Li and Steven C. H. Hoi. *On-Line Portfolio Selection with Moving Average Reversion*. 2012. arXiv: 1206.4626 [cs.CE] (cit. on p. 78).

[54] T.M. Cover. *Universal data compression and portfolio selection*. 1996. DOI: 10.1109/SFCS.1996.548512 (cit. on p. 78).

[55] Emilio Parisotto et al. «Stabilizing Transformers for Reinforcement Learning». In: (2019). arXiv: 1910.06764 [cs.LG] (cit. on pp. 78, 83, 84).

[56] Jason Fernando and Gordon Scott from Investopedia.com. *Bid and Ask*. Mar. 2021. URL: https://www.investopedia.com/terms/b/bid-and-ask.asp (cit. on p. 80).

[57] F. Abdi and A. Ranaldo. *A Simple Estimation of Bid-Ask Spreads from Daily Close, High, and Low Prices*. 2017. DOI: 10.1093/rfs/hhx084 (cit. on p. 81).

[58] G. E. Uhlenbeck and L. S. Ornstein. «On the Theory of the Brownian Motion». In: *Phys. Rev.* 36 (5 Sept. 1930), pp. 823–841. DOI: 10.1103/PhysRev.36.823. URL: https://link.aps.org/doi/10.1103/PhysRev.36.823 (cit. on p. 82).

[59] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG] (cit. on p. 83).

[60] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV] (cit. on p. 86).

[61] «Dynamic Portfolio Selection». In: *Nonparametric Finance*. John Wiley Sons, Ltd, 2018. Chap. 12, pp. 385–418. ISBN: 9781119409137. DOI: https://doi.org/10.1002/9781119409137.ch12. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119409137.ch12. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119409137.ch12 (cit. on p. 93).

[62] «Mean-Variance Portfolio Selection». In: *Robust Equity Portfolio Management + Website*. John Wiley Sons, Ltd, 2015. Chap. 2, pp. 6–21. ISBN: 9781118797358. DOI: https://doi.org/10.1002/9781118797358.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118797358.ch2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118797358.ch2 (cit. on p. 93).

[63] Qingpeng Cai, Aris Filos-Ratsikas, Pingzhong Tang, and Yiwei Zhang. *Reinforcement Mechanism Design for e-commerce*. 2018. arXiv: 1708.07607 [cs.MA] (cit. on p. 98).

[64] Shariq Iqbal and Fei Sha. *Actor-Attention-Critic for Multi-Agent Reinforcement Learning*. 2019. arXiv: 1810.02912 [cs.LG] (cit. on p. 98).

[65] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. Torr, Pushmeet Kohli, and Shimon Whiteson. *Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning*. 2018. arXiv: 1702.08887 [cs.AI] (cit. on p. 98).

[66] Ke Xu, Yifan Zhang, Deheng Ye, Peilin Zhao, and Mingkui Tan. *Relation-Aware Transformer for Portfolio Policy Learning*. July 2020. DOI: 10.24963/ijcai.2020/633 (cit. on pp. 106, 107).